

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**A COMPUTER SIMULATION STUDY
AND COMPONENT EVALUATION FOR A
QUATERNION FILTER FOR SOURCELESS TRACKING
OF HUMAN LIMB SEGMENT MOTION**

by

German A. Henault

March 1997

Thesis Co-Advisors:

Robert B. McGhee
John S. Falby

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED &

19971121 052

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE March 1997		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE A COMPUTER SIMULATION STUDY AND COMPONENT EVALUATION FOR A QUATERNION FILTER FOR SOURCELESS TRACKING OF HUMAN LIMB SEGMENT MOTION			5. FUNDING NUMBERS	
6. AUTHOR(S) Henault, German A.				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/ MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the United States Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Current methods of tracking the human body within virtual environments (VE) are hampered by problems due to interference which occurs from using artificially generated source signals. In recent years, the miniaturization of self-contained inertial tracking systems has made them a viable alternative. They are impervious to external interference but require filtering in order to give accurate orientation data. Filters for this purpose using Euler angles are common, but are limited by their inability to track through the vertical axis. A filter based on quaternions would not have this limitation. This thesis presents an implementation of a quaternion filter in Lisp. The filter was tested with a computer simulated inertial tracker. Also presented is a quantitative and qualitative assessment of an existing inertial tracker, Angularis, which uses a filter based on Euler angles. This effort resulted in an improved filter based on quaternions which allows objects to be tracked through the vertical axis making it a more desirable option for body tracking applications. The evaluation of the Angularis inertial tracker yielded generally good results when tested on a tilt-table at various rates of motion through 45 degrees of rotation. Specifically, orientation errors measured were typically less than one degree for smooth motion. However, when moved rapidly through large orientation angles, it was found that the nonlinear characteristic of the proprietary filter resulted in large steady state errors.				
14. SUBJECT TERMS human interface, virtual environment, articulated humans, human modeling, inertial sensors, quaternions, euler angles, Angularis, InterSense			15. NUMBER OF PAGES 107	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

Approved for public release; distribution is unlimited

**A COMPUTER SIMULATION STUDY AND COMPONENT EVALUATION
FOR A QUATERNION FILTER FOR SOURCELESS TRACKING
OF HUMAN LIMB SEGMENT MOTION**

German A. Henault
Lieutenant, United States Navy
B.A., Temple University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

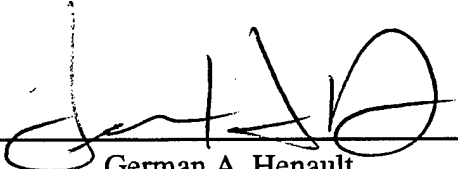
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

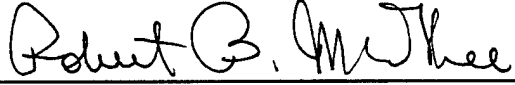
NAVAL POSTGRADUATE SCHOOL

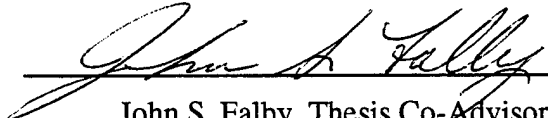
March 1997


Author:


German A. Henault

Approved by:


Robert B. McGhee, Thesis Co-Advisor


John S. Falby, Thesis Co-Advisor


Ted Lewis, Chairman,
Department of Computer Science

from 1915

ABSTRACT

Current methods of tracking the human body within virtual environments (VE) are hampered by problems due to interference which occurs from using artificially generated source signals. In recent years, the miniaturization of self-contained inertial tracking systems has made them a viable alternative. They are impervious to external interference but require filtering in order to give accurate orientation data. Filters for this purpose using Euler angles are common, but are limited by their inability to track through the vertical axis. A filter based on quaternions would not have this limitation.

This thesis presents an implementation of a quaternion filter in Lisp. The filter was tested with a computer simulated inertial tracker. Also presented is a quantitative and qualitative assessment of an existing inertial tracker, Angularis, which uses a filter based on Euler angles.

This effort resulted in an improved filter based on quaternions which allows objects to be tracked through the vertical axis making it a more desirable option for body tracking applications. The evaluation of the Angularis inertial tracker yielded generally good results when tested on a tilt-table at various rates of motion through 45 degrees of rotation. Specifically, orientation errors measured were typically less than one degree for smooth motion. However, when moved rapidly through large orientation angles, it was found that the nonlinear characteristic of the proprietary filter resulted in large steady state errors.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	MOTIVATION.....	1
B.	GOALS	2
C.	ORGANIZATION	3
II.	BACKGROUND	5
A.	MEASURING MOTION TRACKER PERFORMANCE.....	5
B.	TYPES OF MOTION TRACKERS	7
1.	Mechanical.....	7
2.	Electromagnetic	10
3.	Acoustic	11
4.	Inertial	12
C.	RELATED WORK	15
1.	Responsive Workbench TM	15
2.	Computer Graphics System with Force Feedback.....	17
3.	RF/Inertial Head-Tracker.....	17
D.	SUMMARY	18
III.	A QUATERNION ATTITUDE FILTER	19
A.	QUATERNIONS VS. EULER ANGLES	20
1.	Euler Angles.....	20
2.	Quaternions	24
B.	DERIVATION OF QUATERNION FILTER.....	25
C.	SUMMARY	31
IV.	QUATERNION FILTER IN LISP	33
A.	THE SIMULATION MODEL.....	35
B.	SIMULATION RESULTS	38
C.	SUMMARY	42
V.	RESULTS	45

A.	ANGULARIS INERTIAL TRACKING SYSTEM.....	45
B.	QUATERNION FILTER.....	53
VI.	SUMMARY AND CONCLUSIONS	55
A.	SUMMARY	55
B.	CONCLUSIONS.....	56
C.	FUTURE WORK.....	57
	APPENDIX A. DERIVATION OF X MATRIX	59
	APPENDIX B. DERIVATION OF GRADIENT	63
	APPENDIX C. SIMULATION MODEL CODE	65
	APPENDIX D. SIMULATION MODEL TEST RUNS USING GRADIENT DESCENT METHOD	83
	LIST OF REFERENCES	87
	INITIAL DISTRIBUTION LIST	91

LIST OF FIGURES

Figure 1: IPORT Soldier Station [NRG97].	8
Figure 2: Graphics Force Feedback System [HANC96].	9
Figure 3: Virtual, Stereoscopic Displayed Skeleton [GMD97].	16
Figure 4: Projector-Mirror System [GMD97].	16
Figure 5: Coordinate Systems (Frames) [CRAI89].	19
Figure 6: Azimuth, Elevation, and Roll Rotations.	20
Figure 7: Euler Angle Estimation Portion of SANS Filter [MCGH95, BACH96A].	23
Figure 8: Quaternion Filter [BACH96B, MCGH96A].	27
Figure 9: Measured Orientation Vector[BACH96B].	28
Figure 10: Angular Rate Sensor Output [BACH96B].	31
Figure 11: Code Fragment Showing Accelerometer Output Using Quaternions.	36
Figure 12: Deviation and Dip Angle Corrections for the Earth's Local Magnetic Field [FREY96B, MCGH96C].	37
Figure 13: Gradient Convergence Method.	38
Figure 14: Posture Slot Default Value.	39
Figure 15: Gauss-Newton Equation.	41
Figure 16: Gauss-Newton Iteration with 10-degrees Error.	42
Figure 17: Gauss-Newton Iteration with 20-degrees Error.	42
Figure 18: 45-Degree Roll at 10-Degrees/Second.	47
Figure 19: 45-Degree Roll at 45 Degrees/Second.	48
Figure 20: 45-Degree Roll at 90 Degrees/Second.	49

Figure 21: Roll Angle for Random Motions..... 51

Figure 22: Pitch Angle for Random Motions. 52

LIST OF TABLES

Table 1: Evaluation of Position-Tracking Technologies [MEYE92, SKOP96].	13
---	----

ACKNOWLEDGMENTS

I would like to express my sincerest thanks to all the people who have helped me along the way in preparation of this thesis. It's difficult to express, in such a limited number of words, how much my thesis advisor, Dr. Robert McGhee, has changed the way I look at the world around me and how I see myself. His passion for teaching, limitless patience, and dedication to his students has had a profound impact on my life. I will always remember the lessons he has taught me, not only in the classroom but also in life. I would also like to thank my co-advisor, Mr. John Falby, the best C++ programmer I know, a great teacher, and a good friend. His ability to keep us organized and focused when things got crazy was nothing short of amazing. I could not have had two better advisors. Thanks to my fellow classmates, for their friendship and support, from whom I have learned a great deal. Finally, but most importantly, thank you to my wife Diana. Her amazing support through all of my life's endeavors has made it all possible. Thank you for loving me.

I. INTRODUCTION

A. MOTIVATION

Computer systems have experienced a dramatic increase in performance and power in a relatively short period of time. This increased performance makes more realistic and immersive computer simulations possible for research, training and entertainment purposes. However, as rapidly as computer systems have increased in capability, one specific area of computer simulations lags behind. Research into more realistic and natural interactive input devices has come along very slowly and has not had the same success as other aspects of computer hardware. The need for intuitive interaction in virtual environments (VE) has driven the development of several different approaches to accomplishing total immersion into synthetic worlds. The main thrust of research in this realm has been in the area of producing new and improved sensors for tracking the human form and other objects effectively and efficiently and with as little hindrance to the user as possible.

In recent years, inertial sensing technology has become a viable alternative to systems based upon mechanical, electromagnetic, acoustic, and optical tracking sensors. Inertial trackers solve the problems encountered when using these systems, namely, shadowing, metallic, electronic and acoustic interference as well as limitations in range. The self-contained inertial sensors are impervious to any outside interference and function solely by sensing the earth's magnetic and gravitational fields. A combination of accelerometers, angular rate sensors, and magnetometers provide the system with the data required to accurately specify spatial orientations. However, inherent limitations and errors associated

with accelerometers and angular rate sensors necessitates proper filtering be applied in order to extract reliable orientation information.

B. GOALS

A quaternion attitude filter has been proposed which overcomes the singularities encountered when using Euler angles to represent object orientations [MCGH96A]. The goal of this thesis is to implement a simulation of an inertial system that will provide a useful and decisive demonstration of the filter's applicability to body tracking applications. Allegro Common Lisp, ver. 3.0.1 for Windows was chosen as the programming language with which to implement the simulation.

This thesis also takes an existing inertial sensor, the Angularis system, built by InterSense, Inc., [INTE97], and presents a quantitative and qualitative analysis of its performance. Even though this particular sensor was built specifically for the tracking of the human head using a head mounted display (HMD), it is the goal of this study to test the applicability of this particular sensor to other types of tracking. Specifically, to determine the viability of using such a sensor in human limb segment tracking applications. The sensor is evaluated against a system that is very well known and understood by researchers at the Naval Postgraduate School (NPS), the Shallow-Water AUV Navigation System (SANS) which is currently being used in the ongoing Autonomous Underwater Vehicle (AUV) research [BACH96A, MCGH95, MCGH96B]. It is hoped that the Angularis system can perform as well as or better than the much larger SANS system. This advanced, miniaturized inertial sensing technology, used in conjunction with the quaternion filter presented in this thesis, may allow for the creation of a full body suit which would be

capable of tracking an entire human body within VE applications much more reliably and accurately than current systems.

C. ORGANIZATION

Chapter II of this thesis surveys existing sensor technology and presents current work in the area of human body tracking. Chapter III presents a detailed mathematical formulation of a quaternion filter. The quaternion filter is presented as an alternative to the use of Euler angles in representing object orientations. The quaternion representation avoids singularities experienced when tracking objects through the vertical axis as happens when using the more familiar Euler angles. Chapter IV elaborates on the use of the quaternion filter by presenting a simulation model, created as part of this research, to show the viability of using such a filter in conjunction with inertial sensors for object tracking in VE's. Chapter V presents the results of quantitative and qualitative evaluations of the Angularis inertial tracker. Comparisons are made between the Angularis tracker and the SANS system, currently being researched in the NPS AUV project. Chapter VI, the last chapter, presents conclusions about the results of this research and some recommendations for possible future work with quaternion filters and inertial trackers.

II. BACKGROUND

High-performance computer graphics are being applied to an expanding range of domains [BADL93]. One of the most dramatic and exciting areas is the real-time, interactive representation of the human form in training, research and entertainment applications. Several requirements must be met in order to effectively create a realistic representation, namely: 1) create a human model with the desired level of detail, 2) define the level of control and user inputs for manipulating the model, and 3) provide inputs to the model in a timely fashion in order to achieve real-time performance [SKOP96]. Task number two, user input methods, is the focus of this thesis.

The primary purpose of any tracking device is to provide an intuitive interface between human and machine in order to achieve the desired illusion of total immersion. The user must be allowed to interact with the VE in a familiar and natural way. Therefore, the use of standard 2D pointing devices is unacceptable if the goal is to achieve realistic interaction. To that end, a few different types of trackers have been introduced which employ varying methods to capture the position and orientation data of a tracked object [FREY96A]. The remainder of this chapter presents four specific types of trackers as well as quantitative measures by which they can be compared and evaluated.

A. MEASURING MOTION TRACKER PERFORMANCE

Although several different sensor tracking technologies have been developed and applied to VE applications, there exists no standard evaluation method by which to obtain quantifiable comparisons between different types of trackers [SKOP96]. Typically,

financial constraints and the intended application will dictate which tracker is most appropriate for a given project. [MEYE92] suggests some key measures by which tracking systems may be evaluated, namely, (1) *resolution* and *accuracy*, (2) *responsiveness*, (3) *robustness*, (4) *registration*, and (5) *sociability*. Resolution can be defined as the smallest change which can be detected by a given tracking system. The level of detail required by the application will define the resolution required. Higher resolutions are necessary for applications which necessitate the tracking of small, precise movements. Accuracy is the sensor's error range. Given some orientation or position information, the accuracy will determine the range for which the raw data is correct. For example, inertial systems use angular rate sensors which tend to drift over time due to inherent bias errors. The bias determines the accuracy of the sensor and must be accounted for by filtering techniques. A system's *sampling rate*, *data rate*, *update rate*, and *lag* (or *latency*) all combine to describe the overall responsiveness. Sampling rate is simply how often the sensor is checked for new data. The rate at which new data becomes available is the system's data rate. Data rate is defined as the number of computed data points per second that the system can provide. Most systems will implement a much higher sampling rate than data rate in order to assure that new data is not missed. The rate at which the system can provide updated position and orientation data to the host computer is the update rate. This data is raw data and contains errors. Thus, the information must be filtered before it can be used with any degree of reliability. Even then, the accuracy of the position and orientation updates is only as good as the filter. Filtering the sensor data also takes time and will hinder real time updates. A system's lag is sometimes referred to as its latency and is perhaps one of the

most important specifications of a tracking system. Latency is the measure of the delay between the movement of a tracked object and the corresponding movement of the computer representation of that object in the VE. High latency in a tracking system is undesirable. When sufficiently large, it produces visually disturbing anomalies in the computer simulation. These anomalies tend to disorient and confuse participants, possibly even inducing nausea and vomiting [FOX94]. Robustness is the measure of a system's susceptibility to noise and other interference from outside sources. Types of interference include shadowing, metallic, electronic and acoustic. Registration is defined as the coherence between the sensor's actual position and orientation and reported position and orientation. Finally, sociability describes a system's maximum *range of operation*, its *working volume*, and the ability to track multiple targets within that operating range. Working volume is that volume in which the tracker can accurately report position and/or orientation information. [MEY92]

These measures provide a means by which researchers and developers can quantitatively determine the best technological alternative for a given application. However, factors such as availability, cost, and ease of use must also be taken into consideration before making a final decision [SKOP96].

B. TYPES OF MOTION TRACKERS

1. Mechanical

There are generally two different types of mechanical trackers, body-based and ground-based [FREY96B]. Body-based (exo-skeletal) systems are characterized by

interconnected mechanical linkages mounted on the user's body which measure joint angles directly. An example is shown in Figure 1. Since no external source is required, these sensors are not susceptible to external interference and are very accurate. The physical linkages are well suited for providing *haptic responses*. Haptic responses are force feedback cues that enable the user to experience simulated exertion forces during a VE simulation, further enhancing the realism of the environment and immersion of the user.

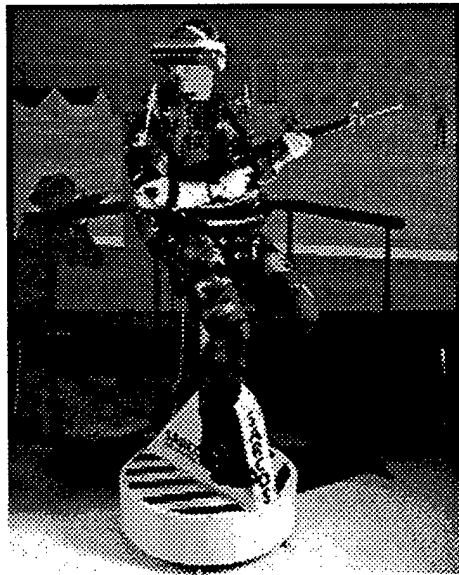


Figure 1: IPORT Soldier Station [NRG97].

The same characteristics, however, that make mechanical tracking devices good for haptic feedback also make such systems cumbersome, heavy, and not very comfortable to use for extended periods of time [FREY96A]. Also, the manner in which the harness is attached to the body must be considered carefully. In order to obtain reliable joint angle data, relative motion between the physical linkages and the human body must be

eliminated. The joints of the device have to remain aligned with the joints of the human user in order to ensure the co-location of their respective centers of rotation [FREY96B].

Ground-based systems are normally not attached to the user. Rather, they track the position and orientation of a separate implement which is manipulated by the user [FREY96B]. These systems, like their body-based counterparts, are very accurate and do not suffer from external interference. However, their applicability is limited by their restrictive working volume. These devices are usually not portable and require a designated area for their use (Figure 2).



Figure 2: Graphics Force Feedback System [HANC96].

In general, mechanical trackers are very precise and responsive to user inputs and are not hindered by external interference. Applications requiring a limited range of motion and where user immobility is not a problem are best suited for this type of sensor. [MEYE92]

2. Electromagnetic

Electromagnetic trackers utilize artificially-generated electromagnetic fields to track position and orientation. A fixed transmitter generates three orthogonal electromagnetic fields which induce voltages in three orthogonal coils located in each detector attached to the tracked object [FREY96B]. These induced voltages are related to the spatial orientation of the detector relative to the transmitter [FREY96B]. This type of system is typically referred to as a *sourced* system with the source being the transmitter which generates the reference electromagnetic fields. Currently, there are two implementations of electromagnetic trackers available, alternating current (ac) and direct current (dc) [MEYE92].

The two implementations work in the same manner except for the way that the reference magnetic field is emitted. The source in an ac system emits continuously changing magnetic fields producing circulating (eddy) currents which in turn produce secondary ac magnetic fields that distort the emitter field pattern [MEYE92]. The dc implementation, on the other hand, emits a sequence of pulses which reduce the effect of distorting currents [MEYE92]. Since eddy currents are created only when the magnetic field is changing, dc systems only generate distortions at the beginning of a measurement cycle [MEYE92]. When the field reaches its steady state, no eddy currents are created, reducing overall system distortion [MEYE92].

Electromagnetic tracking devices are relatively inexpensive and can be used to track numerous objects position and orientation with accuracies adequate for some applications [FREY96B, SKOP96]. The disadvantage, however, is that they have a limited working

volume due to the fact that they are sourced and the magnetic field strength diminishes with distance [MEYE92, SKOP96, POLH93].

3. Acoustic

High frequency ultrasonic sound waves are used to track objects by either the triangulation of several receivers (time-of-flight method, TOF) or by measuring a signal's phase difference between transmitter and receiver (phase-coherence method, PC) [FREY96B, LIPM90]. The TOF method uses the calculated speed of sound through the air to determine the distance between several transmitters and one receiver or vice-versa [SKOP96]. Triangulation formulae are then used, with the calculated distances, to determine object position [LIPM90]. Tracking can be extended to 6-DOF by placing sensors at three separate locations on the same object. Systems utilizing the PC method measure the phase difference between transmitted and received signals and determine the corresponding change in position. However, when an object moves farther than one-half wavelength in a single update period, tracking errors will occur in the position calculation [SKOP96]. Consequently, over time, small errors in position determination will result in large errors overall [FREY96B].

Acoustic systems based on TOF are susceptible to ranging errors due to reduced data rates at greater operating distances. TOF systems are also more vulnerable to spurious noise sources at any range. PC systems are less vulnerable to noise and in general experience improved accuracy, responsiveness, range, and robustness because of their higher data rates, but PC systems are prone to cumulative errors over time. Both systems

must maintain line-of-sight between transmitters and receivers in order to avoid position data errors which result from sensor occlusion (shadowing). [MEYE92]

4. Inertial

Inertial tracking systems use a combination of linear acceleration, angular rate and magnetometer sensors to determine rigid body orientation. Typically, angular orientation is determined by integrating the output from the angular rate sensors [FREY96A]. This is analogous to integrating linear velocity to find position. However, angular rate sensor output is degenerated by an error called *drift*. Drift is defined as the tendency of bias errors, inherent to the sensor, to cause increasing orientation measurement errors over time [FREY96B]. The amount of drift error present in an angular rate sensors output is dependent upon the quality of the sensor, with higher quality sensors having lower bias errors [FREY96B]. This fundamental limitation makes angular rate sensors a short term solution to determining a rigid body's spatial orientation.

In order to compensate for the long term errors introduced by the use of angular rate sensors, inertial systems utilize linear acceleration sensors called accelerometers. Accelerometers measure the gravity vector, in reference to the local coordinate system, as well as forced linear accelerations of the attached rigid body. This confounding of gravity measurement is sometimes referred to as *slosh* [FOX94]. Therefore, if the forced acceleration is \vec{a} , and the acceleration due to gravity is \vec{g} , then the acceleration measured by the accelerometer is $\vec{a}_{\text{measured}} = \vec{a} - \vec{g}$ [FOX94]. Since most real objects do not continuously accelerate, the average of the forced linear acceleration vector will eventually

become zero [FREY96B]. As the average of the forced acceleration vector approaches zero, the averaged accelerometer output will therefore approach $\vec{a}_{\text{measured}} = -\vec{g}$. When averaged over the long term, the accelerometer will produce the gravity vector, expressed in body coordinates, which in turn can be used to calculate the rigid body's pitch and roll angles relative to earth-coordinates [FREY96B]. This observation justifies the use of a *complementary* filter [BROW92] which incorporates the short term accuracy of the angular rate sensors with the long term stability of the accelerometers to provide accurate orientation information for a tracked object.

The third component of inertial systems is the magnetometer. The magnetometer is sensitive to the earth's magnetic field and can sense rotations about the local vertical axis [FREY96B]. Since accelerometers cannot detect rotations about the local vertical, magnetometers must be used to correct drift errors in the azimuth calculations made from angular rate sensor data. Thus, a careful combination of the data from all three sensors can provide a good representation of spatial orientation. A summary of all the sensors presented is given in Table 1.

	Mechanical	Magnetic	Acoustic	Inertial
Accuracy and Resolution	Good.	Good in small working volumes. Accuracy tends to diminish as emitter-sensor distance increases. Accuracy adversely affected by ferromagnetic objects in working volume.	Good.	Good.

Table 1: Evaluation of Position-Tracking Technologies [MEYE92, SKOP96].

	Mechanical	Magnetic	Acoustic	Inertial
Responsiveness	Good.	Relatively low data rates. Filtering required for distortions in emitted field can introduce lag.	TOF: Good at close range. Data rates diminish as range increases. PC: Unaffected by range.	Good.
Robustness	Good. Not sensitive to errors introduced from the environment.	Ferromagnetic objects create eddy currents that distort the emitted field causing ranging errors.	TOF: Low data rates cause vulnerability to ranging errors. Robustness diminishes as range increases and data rates drop. PC: High data rates unaffected by range. Excellent robustness.	Excellent.
Registration	No reports.	No reports.	No reports.	Good.
Sociability	Limited range. Two systems cannot effectively occupy the same working volume.	Most effective for small working volumes. Distortions from induced eddy currents increase with field strength. Configurations available for allowing sensors to share emitters or for multiple emitters in same work space.	TOF: Accuracy and responsiveness diminish as range increases. Small effective working volume can limit sociability. PC: Large working volume offers good sociability. Increased range does not affect responsiveness. Acoustic systems are vulnerable to occlusion.	Excellent.
Comments	Cumbersome. Well suited to force feedback.	Available off-the-shelf. Relatively inexpensive. Most commonly used in current VR research.	Acoustic systems are starting to appear in marketplace and are relatively inexpensive.	Expensive and not widely available.

Table 1: Evaluation of Position-Tracking Technologies [MEYE92, SKOP96].(contd.)

C. RELATED WORK

1. Responsive Workbench™

The Responsive Workbench™ goal is to seamlessly integrate the computer into the user's world [GMD97]. This approach is contrary to the typical VE where the goal is to immerse the user into the computer's world and provide him/her with a virtual presence in that world. With the approach taken by the creators of this system, it is possible for everyday objects and activities to become inputs and outputs to the environment [GMD97]. The display, for instance, is not presented on a traditional computer monitor or television screen but on a real 3D workbench (Figure 3). By doing this, the display becomes part of the human's working environment [GMD97]. Computer-generated stereoscopic images are projected onto a tabletop via a projector-and-mirrors system as illustrated in Figure 4. The 3D effect is observed by wearing shutter glasses. Although currently only one user is tracked, the system allows other participants to observe the 3D interaction with their own shutter glasses. The system also uses a 6-DOF tracking system to track the user's head as well as tracking the user's hands and an input stylus for environment interaction [GMD97].

The creators of the Responsive Workbench™ refer to their system as a "Responsive Environment", which integrates tracking systems, cameras, projectors and microphones, creating a more realistic training and learning environment and challenging traditional expectations of what a computer workstation should be [GMD97]. Current applications of this technology include: medical training, surgical planning, fluid dynamics visualization in a virtual windtunnel, 3D architectural designs, molecular modeling and 3D manipulations of molecular models [GMD97].

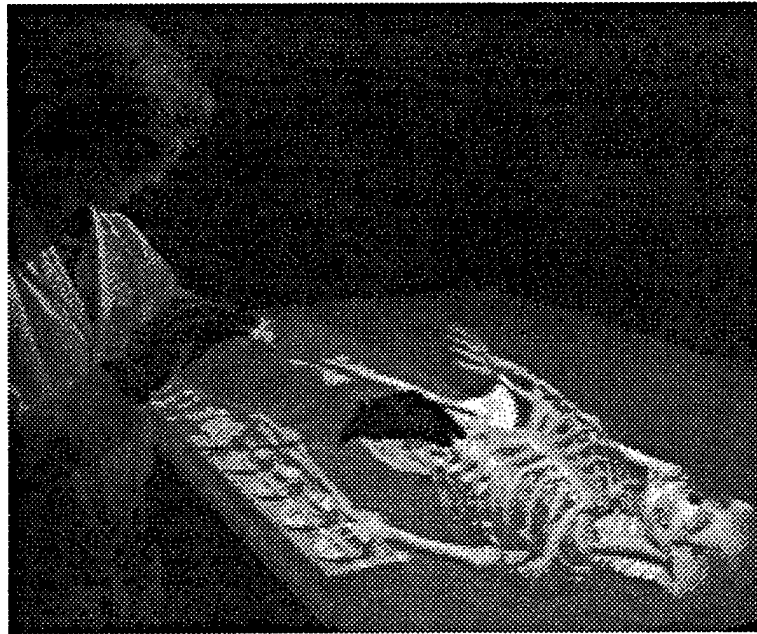


Figure 3: Virtual, Stereoscopic Displayed Skeleton [GMD97].

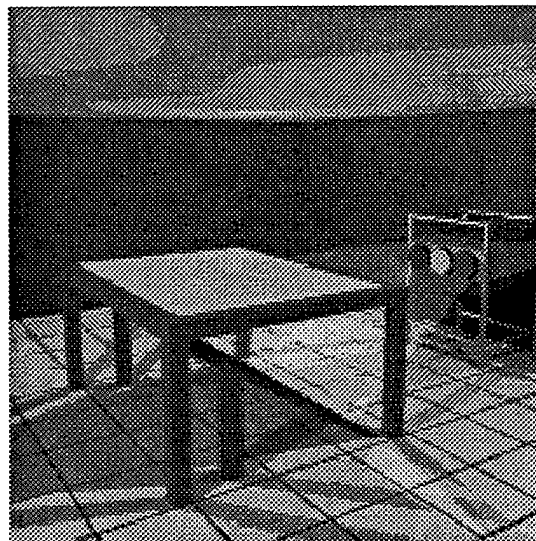


Figure 4: Projector-Mirror System [GMD97].

2. Computer Graphics System with Force Feedback

A more traditional effort, using mechanical sensors, is shown in Figure 2 [HANC96]. This system incorporates both stereoscopic viewing and direct object interaction with a force feedback I/O handheld device [HANC96]. The system allows the user to not only interact with virtual objects within the VE, but to also “feel” them. [HANC96] does not refer to his system as “Virtual Reality” but instead as “Interactive Graphics”. He believes that bimodal displays are the next step in computer graphics and that these systems are not just for virtual reality games, but a serious tool that developers and researchers can utilize in their work.

This system was implemented using two HP 735 UNIX workstations, a Phantom force feedback device, and Crystal Eyes for stereoscopic viewing of the computer monitor [HANC96]. The research focused primarily on the development and implementation of real-time collision detection algorithms which could provide satisfactory user response. The result was a system which could generate stereoscopic rendering rates of 20 Hz, force update rates of over 1000 Hz with an overall system latency of 50 milliseconds [HANC96].

3. RF/Inertial Head-Tracker

A new 3D RF positioning system has been developed at Advanced Position Systems, Inc. (ASPI Technology). Through the use of a new and innovative “dynamic calibration method” developed by Dr. Jun Feng, the 3D RF positioning system improves the positioning accuracy down to the millimeter scale [FENG97]. The success of the initial positioning system prototype has prompted a proposal for a new 6-DOF RF/Inertial tracking system. By incorporating the InterSense 3-DOF inertial tracker [INTE97] for

orientation information, it is proposed that a long range, lightweight 6-DOF cordless head-tracker utilizing RF technology for position information can be developed [FENG97].

D. SUMMARY

This chapter presents a brief overview of available body tracking technology. Table 1 gives a consolidated evaluation summary of the sensors presented with respect to five performance measures. This chapter does not provide a complete review of all available tracking technologies, but rather attempts to present a general overview of this field of technology. Interested readers are directed to [FREY96A, FREY96B, INTE97, MEYE92, POLH93] for further discussion on these and various other systems which are currently available and were not presented here. The next chapter introduces the quaternion filter proposed by [MCGH96A] and presents the complete mathematical formulation of this approach.

III. A QUATERNION ATTITUDE FILTER

In order to completely describe the state of a tracked object, both position and spatial orientation information must be obtained. In robotics, a *coordinate frame* is used to describe the position and orientation of objects with respect to some universal (earth-based) coordinate system, Figure 5 provides an illustration of this idea. The coordinate frame is a local (body) coordinate system which is rigidly attached to an object in a known way [CRAI89]. Calculating the position and orientation of the frame determines the “kinematic” state of the object. Various methods exist for representing the rotations required to bring coordinate systems into alignment. This chapter presents a comparison between the familiar and widely used Euler angle representation for rotations, and the less common quaternion method. Also presented is a mathematical derivation of a quaternion filter, proposed by [MCGH96A] as an alternative to filters based on Euler angles which experience singularities at certain orientations.

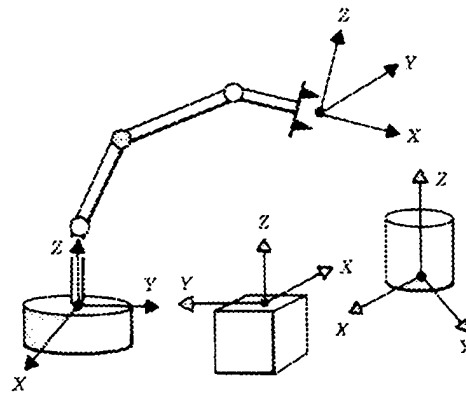


Figure 5: Coordinate Systems (Frames) [CRAI89].

A. QUATERNIONS VS. EULER ANGLES

1. Euler Angles

Orientation information, obtained from the types of motion trackers presented in the previous chapter, is given in local frame coordinates and must be translated into universal coordinates before the object can be graphically represented within a VE. Translating from one coordinate system into another requires the calculation of the rotation(s) and translation(s) which bring both systems into alignment. One of the most popular and intuitive methods of representing the rotation part of such transformations is by the use of Euler angles. The Euler angle method represents the orientation of an object, with respect to a known earth coordinate system, by applying three successive rotations, shown in Figure 6.

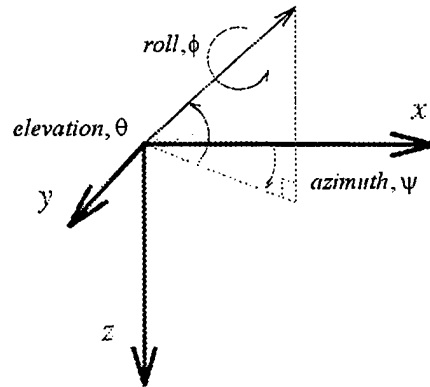


Figure 6: Azimuth, Elevation, and Roll Rotations.

These rotations about the x , y , and z axes are represented by the “elementary” rotation matrices for roll, elevation, and azimuth, respectively, given by [CRAI89]:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \quad (\text{eq. 3.1})$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad (\text{eq. 3.2})$$

$$R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{eq. 3.3})$$

A significant disadvantage of Euler angles occurs when they are used to estimate and calculate orientation angles for tracked objects. A specific example is the navigation filter used in the NPS AUV project [MCGH95, BACH96A]. The filter incorporates inputs from an onboard Inertial Measuring Unit (IMU), heading, and water-speed sensors with intermittent GPS fixes to accurately provide continuous real-time navigational data [BACH96A]. The problem occurs when elevation angles of $\pm 90^\circ$ are encountered. Given the fact that submarines can go vertical only once, this singularity is not important in such navigation problems. However, when applied to human body tracking, singularities can occur because there is nothing to prevent many body segments from passing through a vertical orientation. In a recent research effort, Euler angles were used by [SKOP96] to represent orientations in an implementation of a human body tracking system, using Polhemus™ 3-Space® electro-magnetic tracking sensors [POLH93]. The simulation clearly demonstrated the effects of these singularities when limbs moved through the

vertical axis. [SKOP96] was forced to employ error-checking programming techniques in the software to avoid the problematic orientations and the system crashes that they would have caused when encountering angle singularities.

The reason why elevation angles of $\pm 90^\circ$ are so critical is because Euler angle orientation estimations result in divide-by-zero errors at these angles. A schematic representation of the attitude estimation part of the navigation filter, developed for the NPS AUV project, shown in Figure 7, illustrates this limitation [MCGH95, BACH96A]. As can be seen, the accelerometer estimate of the roll angle, ϕ_a , is determined by using the corresponding value for the elevation angle, θ_a , in the following equations

$$\theta_a = \arcsin \frac{\ddot{x}_a}{g} \quad (\text{eq. 3.4})$$

$$\phi_a = -\arcsin \frac{\ddot{y}_a}{g \cdot \cos \theta_a} \quad (\text{eq. 3.5})$$

Clearly, when θ_a becomes $\pm 90^\circ$, (eq. 3.5) will be undefined due to a divide-by-zero error caused by the fact that $\cos 90^\circ = 0$.

In the same portion of the filter, Euler rates are calculated from body rates using inputs from the angular rate sensors. The angular rates, in body coordinates, are given by the angular rate sensor inputs, (p, q, r) , which are multiplied by the transformation T-matrix and converted into Euler rates as defined below

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi \tan\theta & \cos\phi \tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi \sec\theta & \cos\phi \sec\theta \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (\text{eq. 3.6})$$

Multiplying out (eq. 3.6) yields:

$$\dot{\phi} = p + q \sin\phi \tan\theta + r \cos\phi \tan\theta \quad (\text{eq. 3.7})$$

$$\dot{\theta} = q \cos\phi - r \sin\phi \quad (\text{eq. 3.8})$$

$$\dot{\psi} = q \sin\phi \sec\theta + r \cos\phi \sec\theta \quad (\text{eq. 3.9})$$

Again, elevation angles of $\pm 90^\circ$ create divide-by-zero errors in (eq. 3.7) and (eq. 3.9). In order to avoid such singularities, an alternative representation is needed.

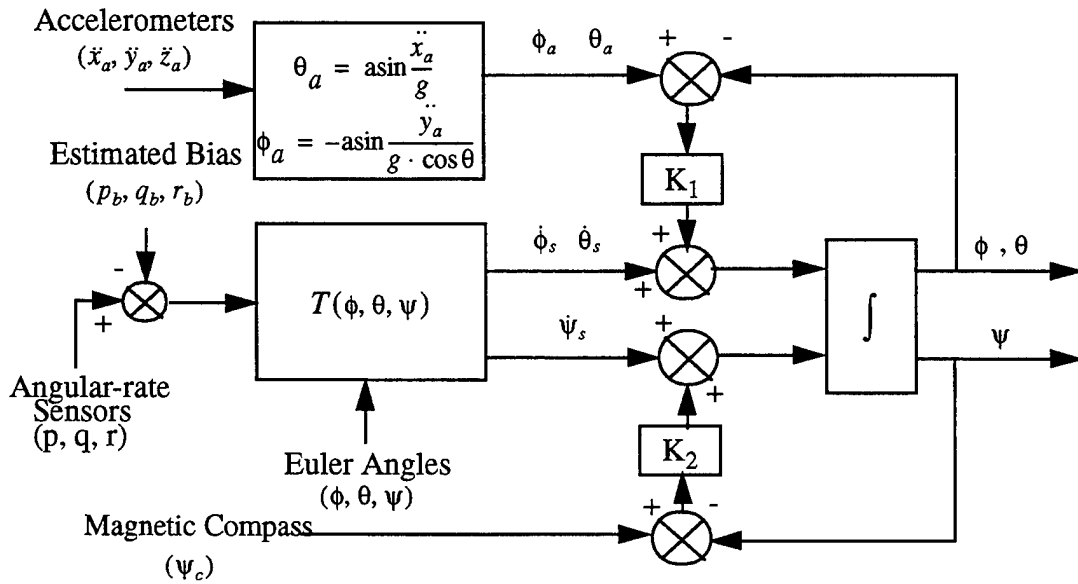


Figure 7: Euler Angle Estimation Portion of SANS Filter [MCGH95, BACH96A].

2. Quaternions

Quaternions are an extension of the familiar complex numbers. Instead of just having one imaginary part, represented by i , quaternions have three “imaginary” parts, represented by i , j , and k , all having the same value, $\sqrt{-1}$. Thus,

$$i * i = i^2 = -1 \quad (\text{eq. 3.10})$$

$$j * j = j^2 = -1 \quad (\text{eq. 3.11})$$

$$k * k = k^2 = -1 \quad (\text{eq. 3.12})$$

Quaternions can also be represented in three different notations. Depending upon the particular application, it may be convenient to represent quaternions as either a linear combination of four components, a four dimensional vector represented by the coefficients of the linear combination, or as a scalar and a vector. That is, the following three notations are equivalent:

$$q = w + xi + yj + zk \quad (\text{eq. 3.13})$$

$$q = (w \ x \ y \ z) \quad (\text{eq. 3.14})$$

$$q = (w, v) \quad (\text{eq. 3.15})$$

When rotating vectors with quaternions, it will generally be required that unit quaternions be used [COOK92]. This is done because of the convenient way in which the inverse of a unit quaternion can be calculated. Specifically, the inverse of a unit quaternion is defined as the conjugate of the quaternion; that is

$$q^{-1} = q^* = (w, -v) \quad (\text{eq. 3.16})$$

where q^{-1} is the inverse of a unit quaternion and q^* is the conjugate of quaternion q .

Any scalar or three dimensional vector can be represented as a quaternion. For scalars, the vector (0 0 0) is appended to the scalar w to obtain

$$q = (w \ 0 \ 0 \ 0) \quad (\text{eq. 3.17})$$

In the case of a three dimensional vector, the scalar 0 is appended to the front of the vector to get the equivalent quaternion.

$$q = (0 \ x \ y \ z) \quad (\text{eq. 3.18})$$

Once scalars and vectors have been properly converted to quaternions, quaternion algebra can be applied to rotate vectors, multiply scalars, etc. Specifically, rotation of a vector, p , by a quaternion, q , is defined as [PAUL90]

$$p_{rotated} = qpq^{-1} \quad (\text{eq. 3.19})$$

where q is a unit quaternion given by

$$q = \left(\cos \frac{\theta}{2}, u \sin \frac{\theta}{2} \right) \quad (\text{eq. 3.20})$$

The symbol u in (eq. 3.20) represents the unit vector about which the vector p is to be rotated through an angle θ . It should be noted, unlike Euler angles, quaternion rotations require only two trigonometric functions to rotate a vector and experience no singularities at any angle of rotation.

B. DERIVATION OF QUATERNION FILTER

Filters are used to minimize errors and return accurate results in an environment where noise corrupts the measurement of the desired data. In the case under consideration, the data is the orientation measurements made by inertial sensors used to track human motion

within a VE. Inertial sensors are inherently noisy and experience measurement errors due to drift, slosh, and various manufacturing imperfections [FOX94, FREY96A]. This section presents a mathematical derivation of the quaternion filter proposed by [MCGH96A] and depicted in Figure 8. This filter was developed as an alternative and improvement to the Euler angle based filter that was designed for the AUV project at NPS [MCGH95, BACH96A].

The quaternion attitude filter takes inputs from three separate sensors which, together, make up the inertial tracking system. The system consists of accelerometers, angular rate sensors, and a 3-axis magnetometer. Accelerometers measure the combination of forced linear accelerations and the reaction force due to gravity, $\vec{a}_{\text{measured}} = \vec{a} - \vec{g}$. Since most real-life objects do not experience constant linear accelerations (i.e. $a \rightarrow 0$), when averaged over time, accelerometers on the average return the gravity vector or local vertical, $\vec{a}_{\text{measured}} = -\vec{g}$ [FOX94, FREY96B]. Angular rate sensors measure the angular velocity in three axes. Output from the angular rate sensors can be integrated to determine position, but because they experience drift errors over time, a combination of accelerometers and magnetometers are required to correct the measured data. Magnetometers measure the earth's magnetic field in body coordinates. The main purpose of the magnetometer triad is to sense the drift error of the angular rate sensors about the vertical axis which can not be sensed by the accelerometers. Together, the three sensor types provide an accurate means of calculating orientation measurements for any object.

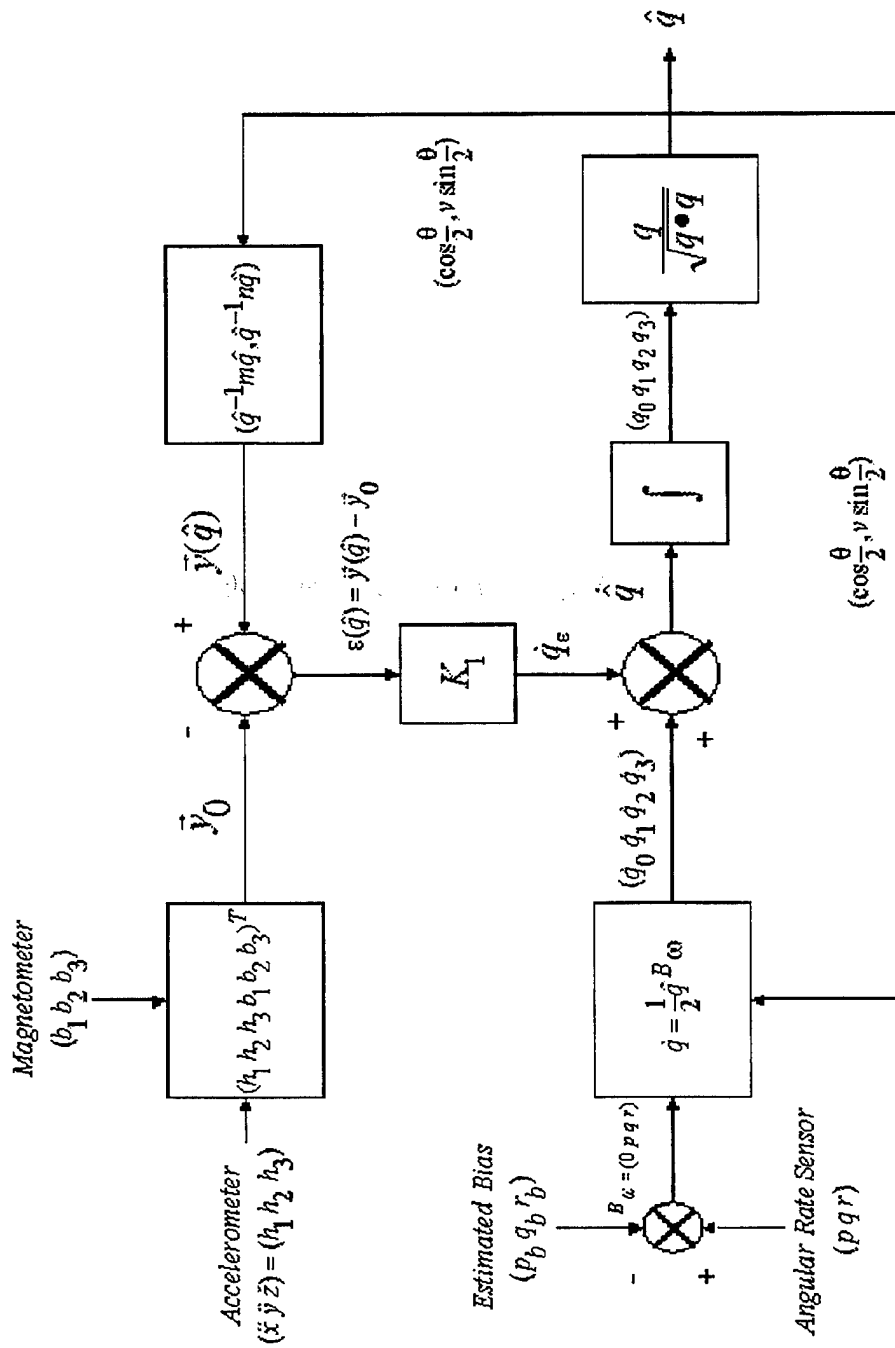


Figure 8: Quaternion Filter [BACH96B, MCGH96A].

The quaternion filter computation begins with the normalized input measurements from the accelerometers and magnetometer, as shown in Figure 9.

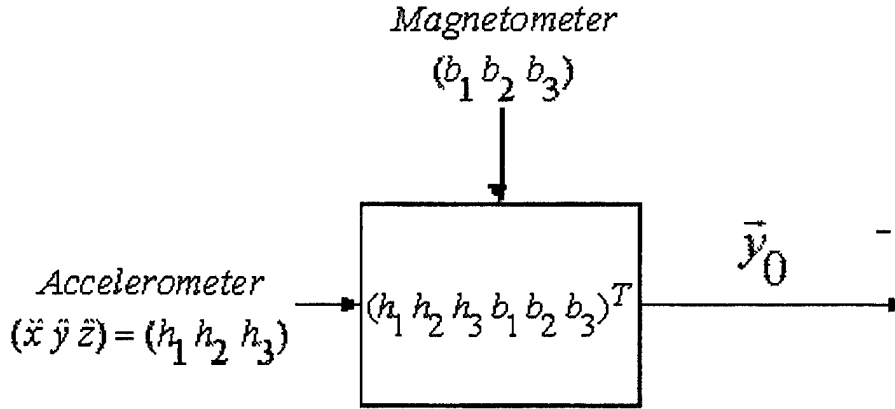


Figure 9: Measured Orientation Vector[BACH96B].

The measurement of the local vertical and the earth's local magnetic field are represented as the 3-dimensional unit vectors given in (eq. 3.21) and (eq. 3.22), respectively.

$$h = (h_1 \ h_2 \ h_3) \quad (\text{eq. 3.21})$$

$$b = (b_1 \ b_2 \ b_3) \quad (\text{eq. 3.22})$$

These vectors are then combined to form the complete measured orientation vector given by

$$y_0 = (h_1 \ h_2 \ h_3 \ b_1 \ b_2 \ b_3) \quad (\text{eq. 3.23})$$

The filter uses the unit orientation quaternion, \hat{q} , which is set when the system is initialized, to calculate the computed measurement vector, $\vec{y}(\hat{q})$. The computed measurement vector is given by

$$\hat{y}(\hat{q}) = (\hat{q}^{-1} m \hat{q}, \hat{q}^{-1} n \hat{q}) \quad (\text{eq. 3.24})$$

where m is the earth's unit gravity vector in earth coordinates and n is the earth's unit magnetic field vector in earth coordinates. Both the m and n vectors are rotated into body coordinates in order to permit comparison with the measured orientation vector given by (eq. 3.23). The error is then computed by taking the difference between the measured vector in (eq. 3.23) and the computed vector in (eq. 3.24), and is given by the expression

$$\epsilon(\hat{q})_{6 \times 1} = \hat{y}(\hat{q})_{6 \times 1} - \hat{y}_{0_{6 \times 1}} \quad (\text{eq. 3.25})$$

One version of the quaternion filter uses an approach known as the "method of steepest descent" to minimize the error in the computed orientation quaternion. The notion of a mathematical gradient is required to implement the method. The gradient is defined as the vector partial derivative of the squared error criterion function which is defined as the square of (eq. 3.25) and is given by

$$\phi(\hat{q})_{1 \times 1} = \epsilon^T(\hat{q})_{1 \times 6} \epsilon(\hat{q})_{6 \times 1} \quad (\text{eq. 3.26})$$

The next step of the filter multiplies (eq. 3.25) by the feedback gain matrix K_1 , which, for this implementation, is defined as

$$K_{1_{4 \times 6}} = K_{4 \times 4} \cdot 2X_{4 \times 6}^T \quad (\text{eq. 3.27})$$

where, for the steepest descent

$$K_{4 \times 4} = -kI_{4 \times 4} \quad (\text{eq. 3.28})$$

and [MCGH67]

$$X_{ij}^T = \left[\frac{\partial y_i}{\partial \hat{q}_j} \right]_{4 \times 6} \quad (\text{eq. 3.29})$$

Appendix A presents the derivation of the X matrix given in (eq. 3.29). The result of the top-half of the filter can then be given by

$$\dot{q}_\epsilon = K_{1_{4 \times 6}} \epsilon(\hat{q})_{6 \times 1} \quad (\text{eq. 3.30})$$

Substituting (eq. 3.27) into (eq. 3.30), gives

$$\dot{q}_\epsilon = K_{4 \times 4} \cdot 2X_{4 \times 6}^T \epsilon(\hat{q})_{6 \times 1} \quad (\text{eq. 3.31})$$

The gradient of the error criterion function given in (eq. 3.26) is defined as

$$\nabla \phi(\hat{q}) = \left(\frac{\partial \phi}{\partial \hat{q}_0}, \frac{\partial \phi}{\partial \hat{q}_1}, \frac{\partial \phi}{\partial \hat{q}_2}, \frac{\partial \phi}{\partial \hat{q}_3} \right)^T = 2X_{4 \times 6}^T \epsilon(\hat{q})_{6 \times 1} \quad (\text{eq. 3.32})$$

Appendix B presents the derivation of the gradient given in (eq. 3.32). Now, substituting (eq. 3.32) into (eq. 3.31), yields

$$\dot{q}_\epsilon = K_{4 \times 4} \nabla \phi(\hat{q})_{4 \times 1} \quad (\text{eq. 3.33})$$

Thus, this new expression gives the output from the top-half of the filter.

The vector given by (eq. 3.33) will be used to correct the computed value for \dot{q} which is calculated using the bias-corrected output of the angular rate sensors in the lower left-hand portion of the filter, Figure 10. The expression for \dot{q} is given by [COOK92]

$$\dot{q} = \frac{1}{2} \hat{q}^B \omega \quad (\text{eq. 3.34})$$

where \hat{q} is the orientation quaternion computed on the previous cycle of the filter and $^B \omega$ is the body coordinate angular rate sensor output, $(p \ q \ r)$, corrected for the estimated

bias, $(p_b \ q_b \ r_b)$. Using (eq. 3.33), the output of the top-half of the filter, to correct (eq. 3.34), gives

$$\dot{\hat{q}} = \dot{q} - K_{4 \times 4} \nabla \phi(\hat{q})_{4 \times 1} \quad (\text{eq. 3.35})$$

The resulting vector from (eq. 3.35) is then numerically integrated and normalized, yielding the next approximation for the orientation quaternion, \hat{q} . The approximation, \hat{q} , can then be used to correct the graphical representation of a tracked object within a VE simulation.

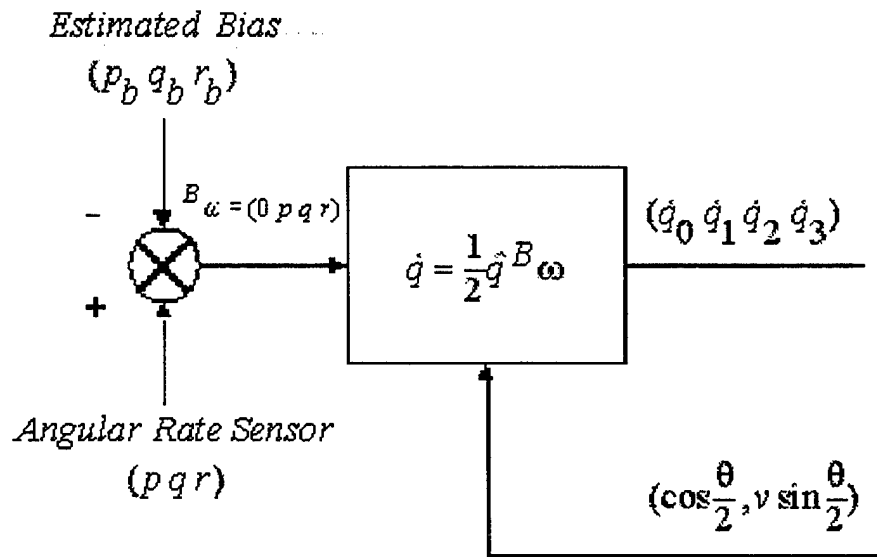


Figure 10: Angular Rate Sensor Output [BACH96B].

C. SUMMARY

The filter presented above offers significant improvements over filters using Euler angles. No singularities exist for any orientations, no trigonometric functions are required, and unit quaternions make calculating inverses a simple matter. This type of filter would be most desirable for applications which track human limb segments, eliminating the

vertical axis singularities which are encountered frequently when Euler angles are used in human motion simulations.

The next chapter presents a Lisp implementation of the quaternion filter. An inertial system, consisting of accelerometers, angular rate sensors, and a 3-axis magnetometer, is simulated to test the convergence and stability of the filter.

IV. QUATERNION FILTER IN LISP

In order to validate the theory presented in the previous chapter, a simulation model was implemented in Lisp. The model simulates output from a static inertial sensor and uses the quaternion filter to correct the resulting orientation quaternion. To simulate error, the filter is initialized to an estimated orientation quaternion which can be offset from the sensor's orientation by as much as 90° . This, of course, is a simulation to evaluate the theory and performance of the filter. One would never expect to see deviations as high as 90° in actual implementations. A deviation that high would suggest a poorly designed filter. Since the inertial sensor remains static throughout the simulation, the user sets the orientation quaternion which represents the orientation of the sensor in space. For the test runs performed in this thesis, an orientation quaternion representing alignment along the x-axis, $(0\ 1\ 0\ 0)$, was chosen. This choice was arbitrary; any quaternion would work just as well. The known orientation quaternion of the sensor is then used to compute the output of the simulated inertial sensors and magnetometer. Using the outputs from the simulated inertial system and the initial orientation quaternion estimate, the filter calculates corrections to the error. If working properly, the filter should respond by gradually converging upon the orientation quaternion of the sensor, in this case $(0\ 1\ 0\ 0)$.

The speed at which the filter converges depends upon the convergence method used. Two different iterative convergence methods were tested in the filter. The "method of steepest descent", utilizing the mathematical notion of a gradient, was used and applied as

described in the previous chapter. Also implemented was the *Gauss-Newton* method [MCGH67] given by

$$\Delta \vec{q} = -\frac{1}{2} \left[X^T X \right]^{-1} \nabla \phi \quad (\text{eq. 4.1})$$

where X represents the same matrix given in (eq. 3.29)

$$X_{ij}^T = \left[\frac{\partial y_i}{\partial \hat{q}_j} \right]_{4 \times 6}$$

and $\nabla \phi$ is the gradient given by (eq. 3.32)

$$\nabla \phi(\hat{q}) = \left(\frac{\partial \phi}{\partial \hat{q}_0}, \frac{\partial \phi}{\partial \hat{q}_1}, \frac{\partial \phi}{\partial \hat{q}_2}, \frac{\partial \phi}{\partial \hat{q}_3} \right)^T = 2X_{4 \times 6}^T \epsilon(\hat{q})_{6 \times 1}$$

Substituting (eq. 3.32) into (eq. 4.1) yields

$$\Delta \vec{q}_{4 \times 1} = - \left[X^T X \right]_{4 \times 4}^{-1} X_{4 \times 6}^T \epsilon(\hat{q})_{6 \times 1} \quad (\text{eq. 4.2})$$

When using the Gauss-Newton method, (eq. 4.2) represents the output from the top-half of the filter and is used to correct the rate output from the angular rate sensor in the lower left-hand portion of Figure 8. It should be noted, however, that the form of the Gauss-Newton equation in (eq. 4.2), with a scalar multiplier of -1, is only applied to noiseless, perfect data. That is, (eq. 4.2) treats accelerometer and magnetometer data as if they were perfect measurements of m and n , the gravity and magnetic field vectors, respectively. Since in the real world this is hardly ever the case, when dealing with data corrupted by noise, a scalar multiplier α is used, defined as

$$\alpha = k \Delta t \quad (\text{eq. 4.3})$$

where $0 < \alpha < 1$. The resulting numerical integration equation becomes

$$\hat{q}_{n+1} = \hat{q}_n + \frac{1}{2} \hat{q}_n^B \omega \Delta t + \alpha [X^T X]^{-1} X^T \epsilon(\hat{q}_n) \quad (\text{eq. 4.4})$$

yielding the next approximation for the orientation quaternion.

A. THE SIMULATION MODEL

The quaternion filter takes five inputs, namely, accelerometer, angular rate, and magnetometer sensor output describing measured orientation data, a delta-t, and the last computed value of the orientation quaternion, \hat{q} . Accelerometer output is defined as a vector of three components, $(\ddot{x}_a, \ddot{y}_a, \ddot{z}_a)$, given by [MCGH96B]

$$\ddot{x}_a = \dot{u} + qw - rv + g \sin \theta \quad (\text{eq. 4.5})$$

$$\ddot{y}_a = \dot{v} + ur - pw - g \sin \phi \cos \theta \quad (\text{eq. 4.6})$$

$$\ddot{z}_a = \dot{w} + pv - uq - g \cos \phi \cos \theta \quad (\text{eq. 4.7})$$

To remain a purely quaternion implementation, the Euler angle representations for the gravity vector components in (eqs. 4.5, 4.6, 4.7) are replaced by equivalent quaternion expressions. This is done by rotating the earth-based gravity vector, m , into body coordinates using the orientation quaternion of the inertial sensor. The resulting quaternion representation for the gravity vector, in body coordinates, is analogous to the $g \sin \theta, -g \sin \phi \cos \theta, -g \cos \phi \cos \theta$ terms above. The Lisp implementation of the quaternion representation for (eqs. 4.5, 4.6, 4.7) is shown in Figure 11, taken from the take-accelerometer-reading method of the inertial-sensor class located in the file sensor.lsp in Appendix C.

```

(body-gravity-vector (rotate-vector
                     (quaternion-inverse (orientation-quaternion
                                           inertial-sensor)) *m*))

(ax (+ u-dot (* w q) (* -1 v r) (second body-gravity-vector)))
(ay (+ v-dot (* u r) (* -1 w p) (third body-gravity-vector)))
(az (+ w-dot (* v p) (* -1 u q) (fourth body-gravity-vector)))

```

Figure 11: Code Fragment Showing Accelerometer Output Using Quaternions.

Note that (eq. 3.19)

$$p_{rotated} = qpq^{-1}$$

is a rotation of a vector, p , from body coordinates to earth coordinates. Since the gravity vector is rotated from earth coordinates to body coordinates, the inverse of (eq. 3.19) is applied, namely,

$$m_{body} = q^{-1} m_{earth} q \quad (\text{eq. 4.8})$$

This is done by using the function *rotate-vector()*. *rotate-vector()* takes a unit quaternion and an arbitrary vector and applies the quaternion rotation in (eq. 3.19). To effect the reverse rotation, as is required for the gravity vector, the inverse of the orientation quaternion is passed to *rotate-vector()* as shown in Figure 11. The resulting rotation is the desired earth-to-body vector rotation given in (eq. 4.8).

Magnetometer output is generated by a magnetometer class [FREY96B, MCGH96C]. Analogous to the simulated accelerometer reading, simulation of the magnetometer requires that the earth's magnetic field vector, n , expressed in earth coordinates, be rotated to body coordinates. Before this can be done, however, the magnetometer simulation

incorporates the *deviation* and *dip angle* [FREY96B] characteristics of the earth's magnetic field in the local Monterey, California area. Magnetic deviation is defined as the difference between the north compass heading and the true geographic north at a given location on the earth's surface. Monterey requires a correction of 15° [FREY96B, MCGH96C]. The lines of earth's magnetic force are parallel to the surface at the earth's equator. However, as the lines approach the magnetic poles, they become increasingly vertical. Dip angle, is the correction for the measure of the local downward deflection of the magnetic field. In Monterey, a correction of -60° is applied [FREY96B, MCGH96C]. The correction for the deviation and dip angle [FREY96B, MCGH96C] to the magnetic field vector is implemented in the code fragment shown in Figure 12.

```
(defun earth-magnetic-field-unit-vector (deviation dip-angle)
  (rest (rotate-vector (equivalent-quaternion deviation (- dip-angle) 0)
    '(0 1 0 0))))

;normalized earth magnetic field vector in earth coordinates for Monterey, CA.
(setf *n* (cons 0
  (earth-magnetic-field-unit-vector (deg-to-rad 15) (deg-to-rad 60))))
```

Figure 12: Deviation and Dip Angle Corrections for the Earth's Local Magnetic Field [FREY96B, MCGH96C].

Once corrected, the normalized magnetic field vector, n , is rotated to body coordinates using the rigid body's orientation quaternion, similar to the gravity vector, m , in the accelerometer simulation. Angular rate sensor output, for this implementation, was kept at $(0\ 0\ 0)$, since the inertial sensor remains static throughout the simulation. Together, the values calculated for each sensor simulates the output from an inertial sensor at rest and constitutes the measured orientation vector in the quaternion filter.

B. SIMULATION RESULTS

Each convergence method has its own test-filter function used to begin the simulation. The user provides values for k (gradient method), or multiplier (Gauss-Newton method), the desired initial offset of the estimated quaternion from the orientation of the sensor, maximum number of iterations, an error tolerance, and a Δt for the Euler integration portion of the filter. The test-filter function for the gradient method is shown below in Figure 13.

```
(defun test-filter-gradient (k-value offset iterations error-threshold delta-t)
  (setf *k* k-value)
  (setf Sensor (make-instance 'inertial-sensor))
  (initialize Sensor)
  (calibrate-magnetometer Sensor -1 1 -1 1 -1 1)
  (let* ((accelerometer (take-accelerometer-reading Sensor))
         (magnetometer (take-magnetometer-reading Sensor))
         (angular-rate (take-angular-rate-sensor-reading Sensor))
         (q-hat offset)
         (delta-t delta-t)
         (error 1))
    (do ((x 1 (1+ x)))
        ((> x iterations)
         (format t "~%~%***** Iteration ~A *****~%" x)
         (if (>= error error-threshold)
             (setf output (quaternion-filter-gradient accelerometer
                                                         magnetometer
                                                         angular-rate
                                                         q-hat
                                                         delta-t)
                    q-hat (firstn 4 output)
                    error (error-difference (lastn 6 output)))
             (setf x (1+ iterations))))))
```

Figure 13: Gradient Convergence Method.

The function instantiates an instance of an inertial sensor, which has a super class of quaternion-rigid-body. Associated with the new inertial sensor object is a slot, inherited from quaternion-rigid-body, called *posture*. Posture is a vector containing both position

and orientation information, (xe ye ze q0 q1 q2 q3), for the object instance. The default posture is set to (0 0 0 0 1 0 0), corresponding to a position at the origin of the earth coordinate system and an orientation quaternion of (0 1 0 0), as shown in Figure 14 below.

```
(defclass quaternion-rigid-body ()
  (
    ;the vector (xe ye ze q0 q1 q2 q3).
    (posture
      :initform '(0 0 0 0 1 0 0)
      :initarg :posture
      :accessor posture)
```

Figure 14: Posture Slot Default Value.

The quaternion-rigid-body class also has a slot called *orientation-quaternion*. Orientation-quaternion contains the last four values of the slot posture, corresponding to the quaternion portion of the rigid body state. This slot is then used to rotate all vectors within the filter implementation.

After the test function initializes the sensor and calibrates the magnetometer, readings from each of the three sensors of the inertial tracking system are taken. These values are assigned to variables corresponding to each of the three different sensors. This is done only once in this implementation of the simulation because the sensor remains static. Therefore, the posture of the sensor does not change, requiring no updates to the filter be made. A real system would need to take readings from the individual sensors on every iteration to correct for varying orientations of a moving sensor package.

The test function then enters a loop which calls the appropriate version of *quaternion-filter()* until one of two conditions are met. The loop can either terminate after completing the maximum number of iterations, whether or not the filter has converged, or when the orientation-quaternion estimation is within the specified error-tolerance. The values of all

critical variables are printed on every iteration to allow for easy evaluation of filter performance.

As mentioned above, two convergence methods were tested in the implementation of the quaternion filter. The gradient method, presented in Chapter 3, guarantees that a linear convergence to a local minimum will occur given a sufficiently small value for k . The value of k represents the size of the step taken along the gradient vector towards the minimum value, where the gradient vector lies perpendicular to the contour lines of a given surface. Large values for k mean large steps with faster convergence, but the possibility of overshooting the local minimum. Small values for k mean smaller steps with slower convergence, and a reduced chance of overshooting. An optimal value for k can be chosen using a method called the Newton-Raphson iteration [MCGH67]. This method was not used in this simulation and is beyond the scope of this thesis. Filter performance, using the gradient method, was good. The filter was given a fairly large error. Specifically, an offset of 20° from the orientation quaternion of $(0 \ 1 \ 0 \ 0)$, given by $(0 \ .939692621 \ .342020143 \ 0)$, was used to initialize the filter. The filter converged in an acceptable number of iterations with k values in the range $0.7 \leq k \leq 1.2$, averaging 22 iterations, with the best performance at $k = 1.2$, 17 iterations. Using an error of 10° , $(0 \ .9848077530 \ .173648177 \ 0)$, yielded better results, as expected. The same range of k values averaged 15 iterations to converge with $k = 1.2$ being the best, converging in 12 iterations. The complete test runs are included in Appendix D.

Test runs using the Gauss-Newton method well out-performed the gradient method. Gauss-Newton converges quadratically [MCGH67], often requiring only one iteration for small errors. Test runs, similar to the gradient method, were performed using the perfect sensor multiplier of -1. It should be noted that the call to *test-filter-gauss-newton()* is made with the multiplier set to 10 vice 1. Referring to (eq. 4.3)

$$\alpha = k\Delta t$$

the multiplier α is expressed as the product of a constant, k , and Δt . Since delta-t is treated as a separate variable in the code, set to 0.1 in this simulation, the value given to the multiplier should cause (eq. 4.3) to equal 1 when performing the numerical integration in (eq. 4.4)

$$\hat{q}_{n+1} = \hat{q}_n + \frac{1}{2}\hat{q}_n^B \omega \Delta t + \alpha [X^T X]^{-1} X^T \varepsilon(\hat{q}_n)$$

Therefore, a value of 10 for the variable *multiplier* yields the correct value for (eq. 4.3).

Also, α is actually negative, but this is taken care of in the code, when calculating the value of (eq. 4.2)

$$\Delta \vec{q}_{4 \times 1} = -[X^T X]_{4 \times 4}^{-1} X_{4 \times 6}^T \varepsilon(\hat{q})_{6 \times 1}$$

as shown in Figure 15

(delta-q (scalar-multiply-vector (* -1 multiplier)
(post-multiply X-squared-inv (post-multiply X-trans error))))

Figure 15: Gauss-Newton Equation.

This was done in order to simplify the function call, avoiding inadvertent calls with positive values for the multiplier. Using the same 10° and 20° errors for the initial orientation quaternion estimation as the gradient method test runs, the Gauss-Newton method gave the following results, shown in Figure 16 and Figure 17.

```
(test-filter-gauss-newton 10 10-degrees 10 .001 .1)
initial q-hat (0 0.984807753 0.173648177 0)

***** Iteration 1 *****
new-q-hat (5.62901044742602E-4 0.999968870927989 0.0045325612552949 -
0.00643398833417277)
***** Iteration 2 *****
new-q-hat (-2.78220979546233E-5 0.99999999558864 -6.85807994618875E-6 -
7.82114313870801E-6)
```

Figure 16: Gauss-Newton Iteration with 10-degrees Error

```
(test-filter-gauss-newton 10 20-degrees 10 .001 .1)
initial q-hat (0 0.939692621 0.342020143 0)

***** Iteration 1 *****
new-q-hat (-0.00212904587800956 0.999131190052156 0.0337658829594332 -
0.0243351058469248)
***** Iteration 2 *****
new-q-hat (-7.57124440093844E-4 0.999999679561069 -1.35855048017928E-4 -
2.21774091515007E-4)
***** Iteration 3 *****
new-q-hat (-1.06750499371424E-8 0.999999999999999 -1.43682207139414E-8
4.92518736799743E-8)
```

Figure 17: Gauss-Newton Iteration with 20-degrees Error.

C. SUMMARY

The theory presented in Chapter 3 has been shown to be sound, as evidenced by the results obtained in the test runs presented. Two methods of convergence were successfully implemented. The gradient method showed acceptable results, converging quickly for large errors in the initial estimation of the orientation quaternion. Gauss-Newton, on the other hand, resulted in quadratic convergence, requiring only 9 iterations to converge for

an initial error of 89° ! Given the initial test results, it can be concluded that, the quaternion filter is a viable alternative to Euler angle based filters, especially in the realm of human body tracking. The errors used in the test runs were much larger than should ever be encountered in an actual inertial tracking system implementation. The quaternion filter, given a reasonably accurate inertial tracking system, should never allow the orientation quaternion to be off as much as was shown in this thesis. The next chapter presents the results of quantitative and qualitative analysis done on an existing inertial tracking system, the Angularis inertial tracker, manufactured by InterSense, Inc. [INTE97].

V. RESULTS

A. ANGULARIS INERTIAL TRACKING SYSTEM

The Angularis tracker is a palm-sized plastic block which contains three orthogonal angular rate sensors, three orthogonal accelerometers, and a two-axis magnetometer to determine the angular orientation of the human head [FREY96A]. Built initially for HMD applications, the extensibility of the Angularis to entire body tracking is quickly becoming a possibility. Inertial systems, like the Angularis, do not suffer from the traditional shortcomings of sensors based on mechanical, electromagnetic and acoustic technologies. They are self-contained, requiring no outside source to calculate spatial orientation, making them immune to outside interference and extending their range beyond that of any traditional tracking system. Until recently, the only limitation to applying inertial navigation technology to body tracking was the fact that inertial sensors were big, bulky unwearable devices. However, with the advent of state-of-the-art, micro-machined inertial sensor components, this concern is no longer a factor.

The question being investigated is whether or not the Angularis, in its present configuration, would be applicable to human body limb segment tracking within a VE simulation. Since the sensor was originally built to be worn as a head tracking device, inherent assumptions in the design of the sensor and filtering technique could possibly prove to be a limiting factor in extending its use to entire body tracking.

The system consists of the inertial tracker and a computer processing unit which receives data from the sensor, processes it, and delivers filtered orientation data to the host computer via an RS-232 cable connection. In its present configuration, data from the

individual sensors, within the plastic block, is inaccessible. A “hard-coded” Kalman filtering technique is used to filter data from the sensor [INTE96]. The filter is “hard-coded” in the sense that it is impossible to apply any outside filter, such as the quaternion filter, to the sensor output. The only choice given to the user is between full-order or reduced-order Kalman filtering [INTE96]. Reduced-order is the default, allowing the system to run above 500 Hz and delivering almost the same performance as full-order Kalman filtering [INTE96]. Once the filtered data is passed to the host computer, VE software running on the host can use the orientation data to update graphical representations of tracked objects.

In order to test the performance of the Angularis sensor, mechanical tilt-table tests were performed at varying rates of rotation. The sensor was strapped onto the tilt-table and allowed to measure the roll angle as the unit was rotated through 45° . The roll was performed two times at each rate, allowing the sensor to stabilize at the end of each roll for 20 seconds. The results are shown in Figure 18, Figure 19, and Figure 20. The Angularis performed well, evidenced by the outputs shown on the graphs. However, the sensor does not correct for errors at the end of the roll as in the SANS filter tests by [BACH96A, ROBE97]. Specifically, while the SANS filter graphs show a gradual correction being applied to the angle as the system stabilizes at the end of the roll, no such corrections are made by the Angularis. Whatever angle is returned at the end of the roll, no matter what the error, the sensor “locks on” to that reading and does not correct. This is shown by the flat line at the end of each roll in the graphs. This lack of immediate correction for the error

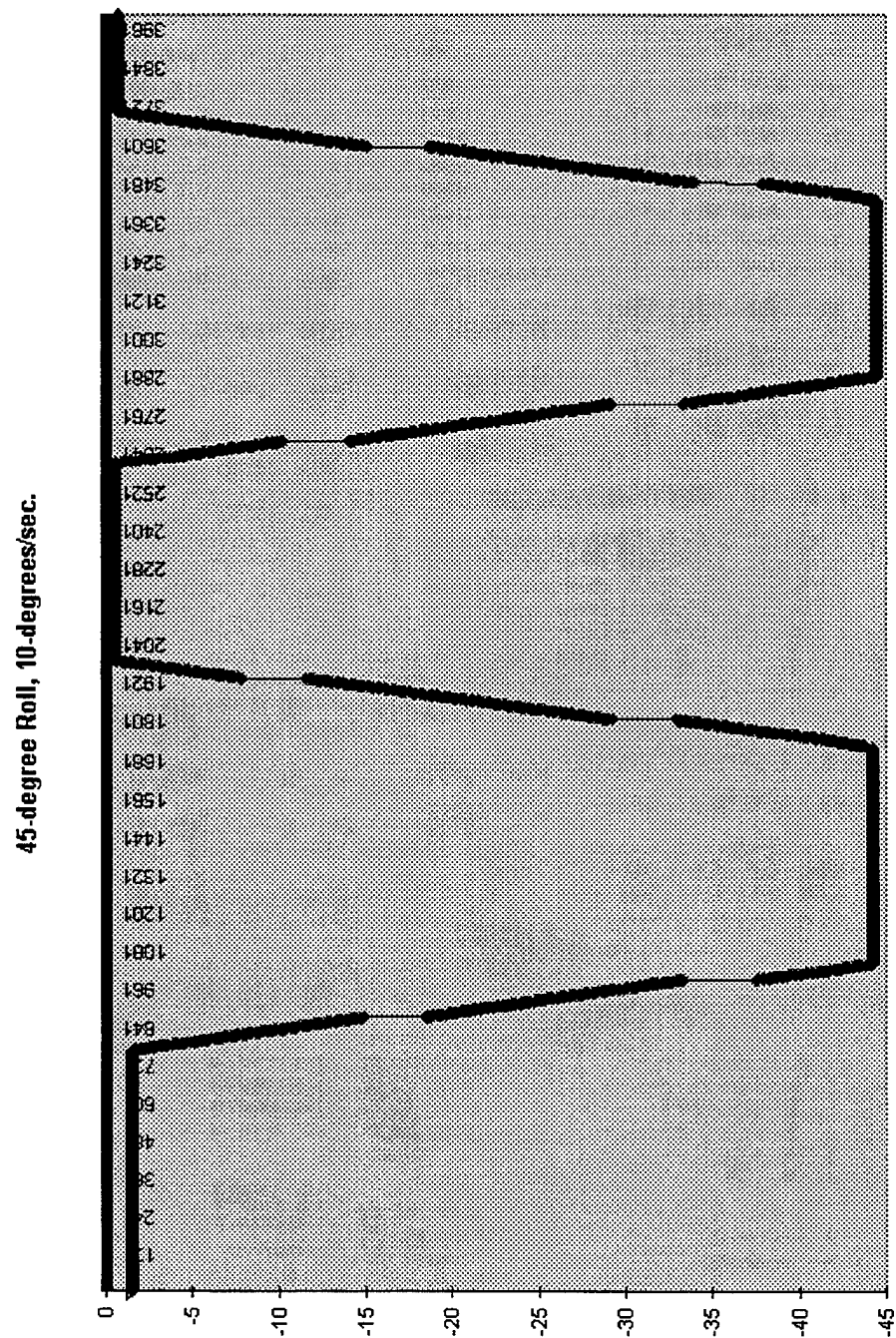


Figure 18: 45-Degree Roll at 10-Degrees/Second.

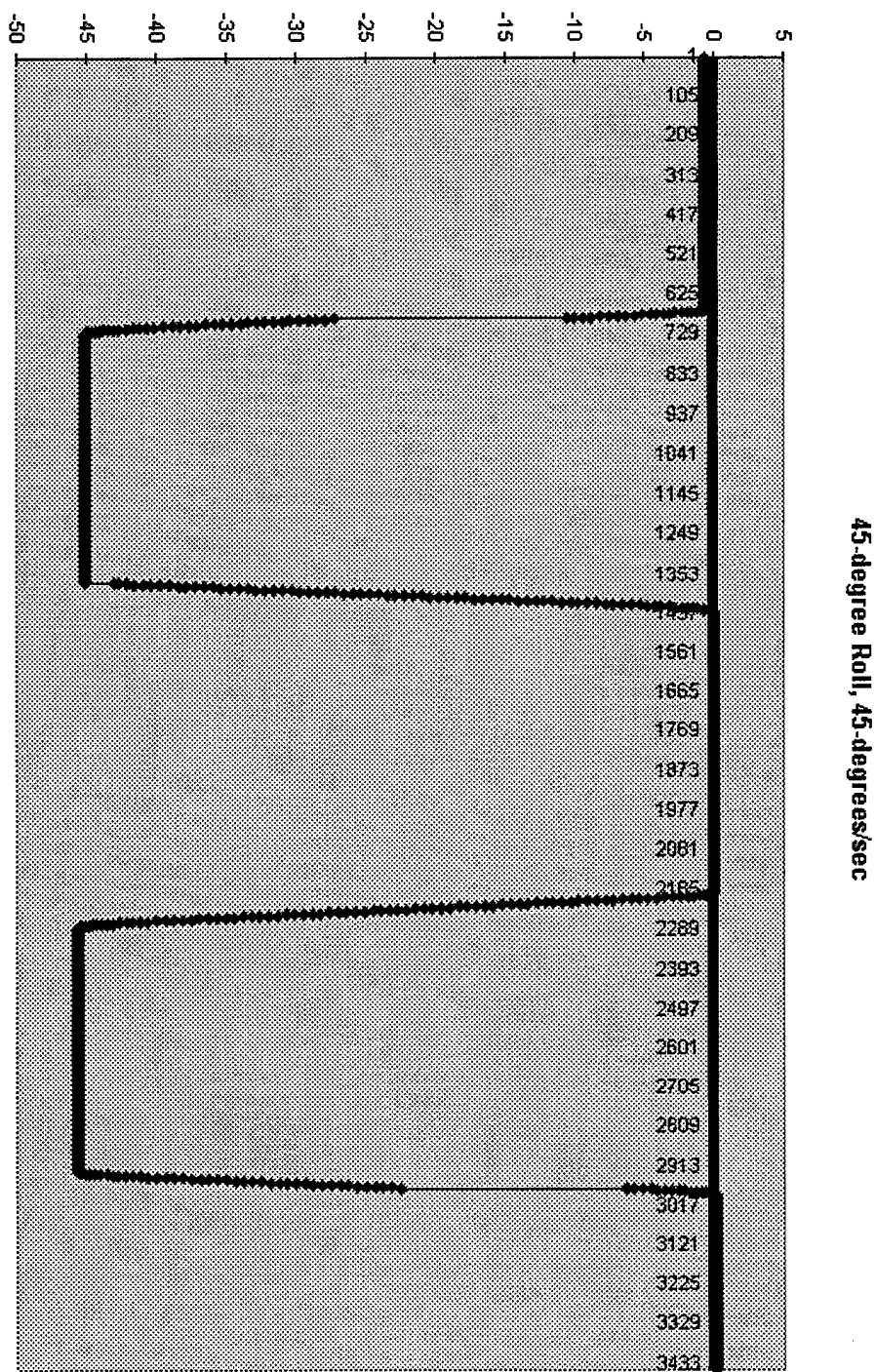


Figure 19: 45-Degree Roll at 45 Degrees/Second.

45-degree Roll, 90-degrees/sec

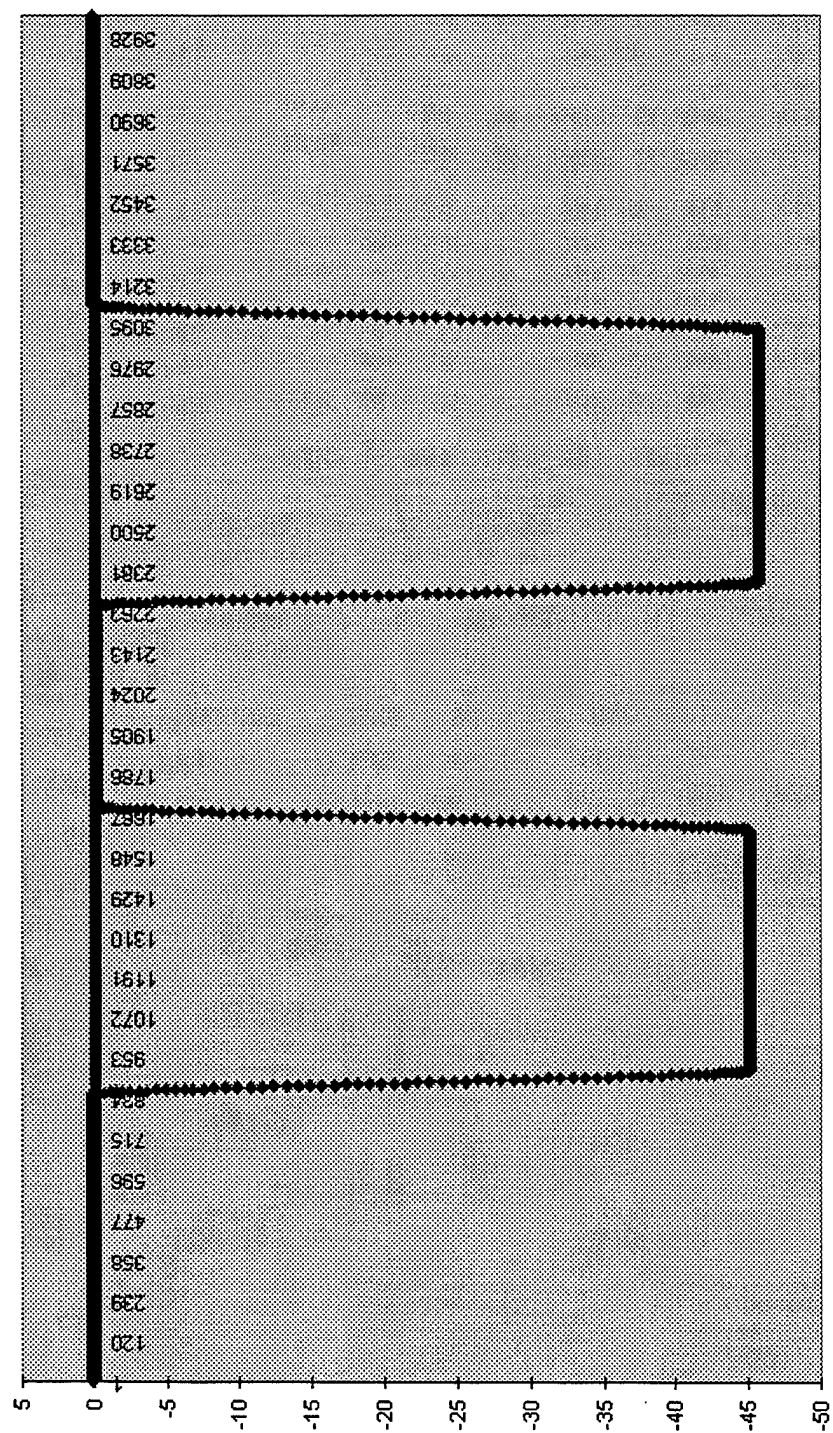


Figure 20: 45-Degree Roll at 90 Degrees/Second.

may be attributed to the fact that the sensor was originally designed for HMD applications. In his paper, [FOX94] explains the drift correction method applied to the first prototype of the Angularis sensor. Using the heuristic that the human head pauses every 10 seconds in a typical HMD simulation, [FOX94] explains that this pause would allow the fluid-filled inclinometer to settle to its correct pitch and roll in approximately 1/4 second. When this occurred, the orientation values would be corrected and reset. However, the error would not be corrected all at once. Gradual correction of the error was done to prevent jarring the user, a real concern with HMD applications [FOX94]. This design feature may explain the output shown in the graphs above.

The fact that individual sensor outputs, from the components of the inertial system, are not accessible makes the overall system limited in its application. It is doubtful that body tracking applications would fare well with the current configuration. A qualitative test was performed using the software provided by the manufacturer. The software represented the sensor block as a virtual cube which moved in response to the movements applied to the actual sensor. It was observed that the graphical representation would not correspond to movements applied to the sensor after just a few arbitrary rotations. When the sensor was placed flat on the table, the graphical representation manifested the errors which had accumulated. This error could be corrected by jiggling the sensor until the graphical representation matched the orientation of the actual sensor. This anomaly may be able to be corrected if the system had allowed the application of an alternative filter. Graphs representing the output from random rotations are shown in Figure 21 and Figure 22. The

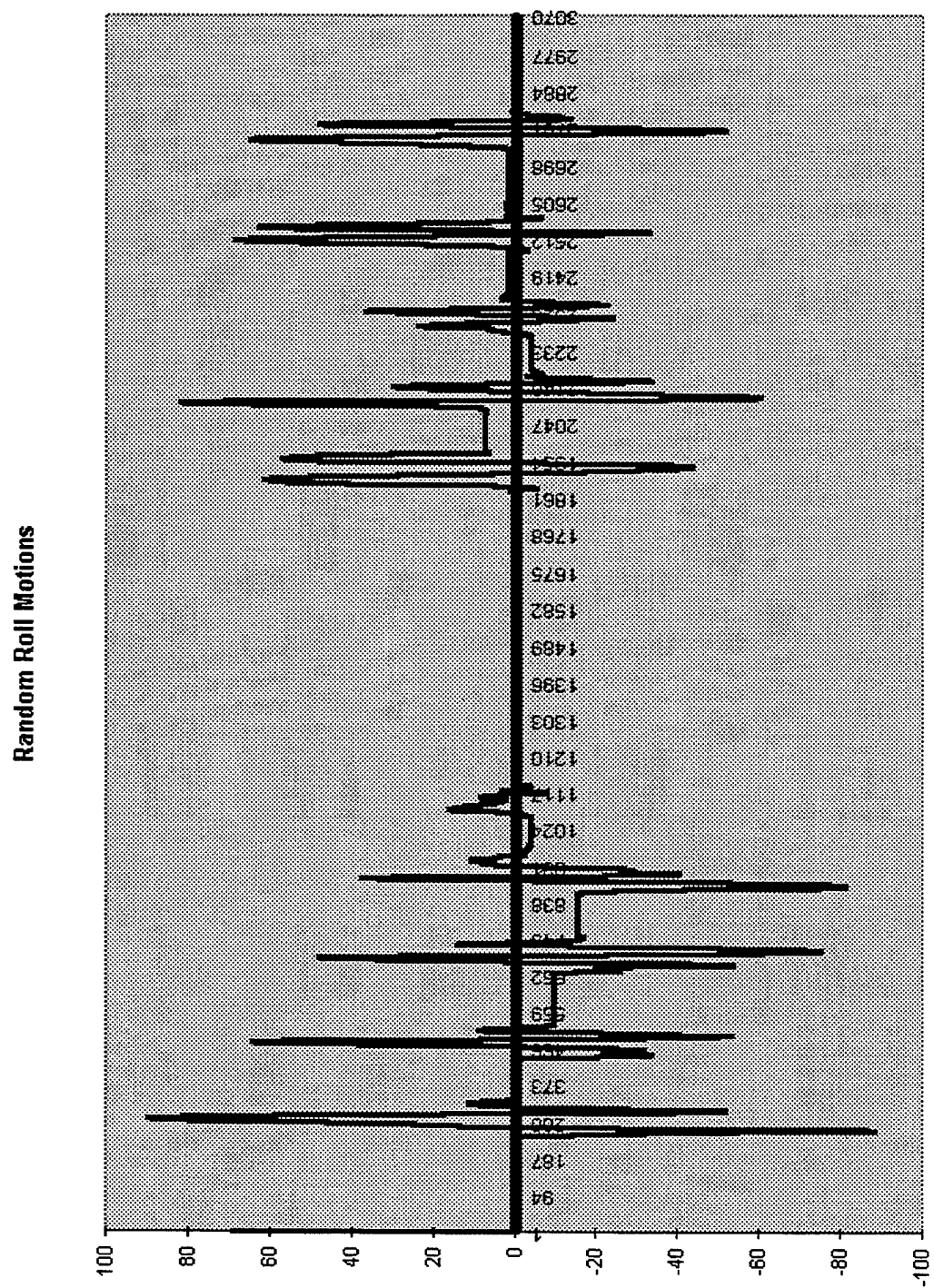


Figure 21: Roll Angle for Random Motions.

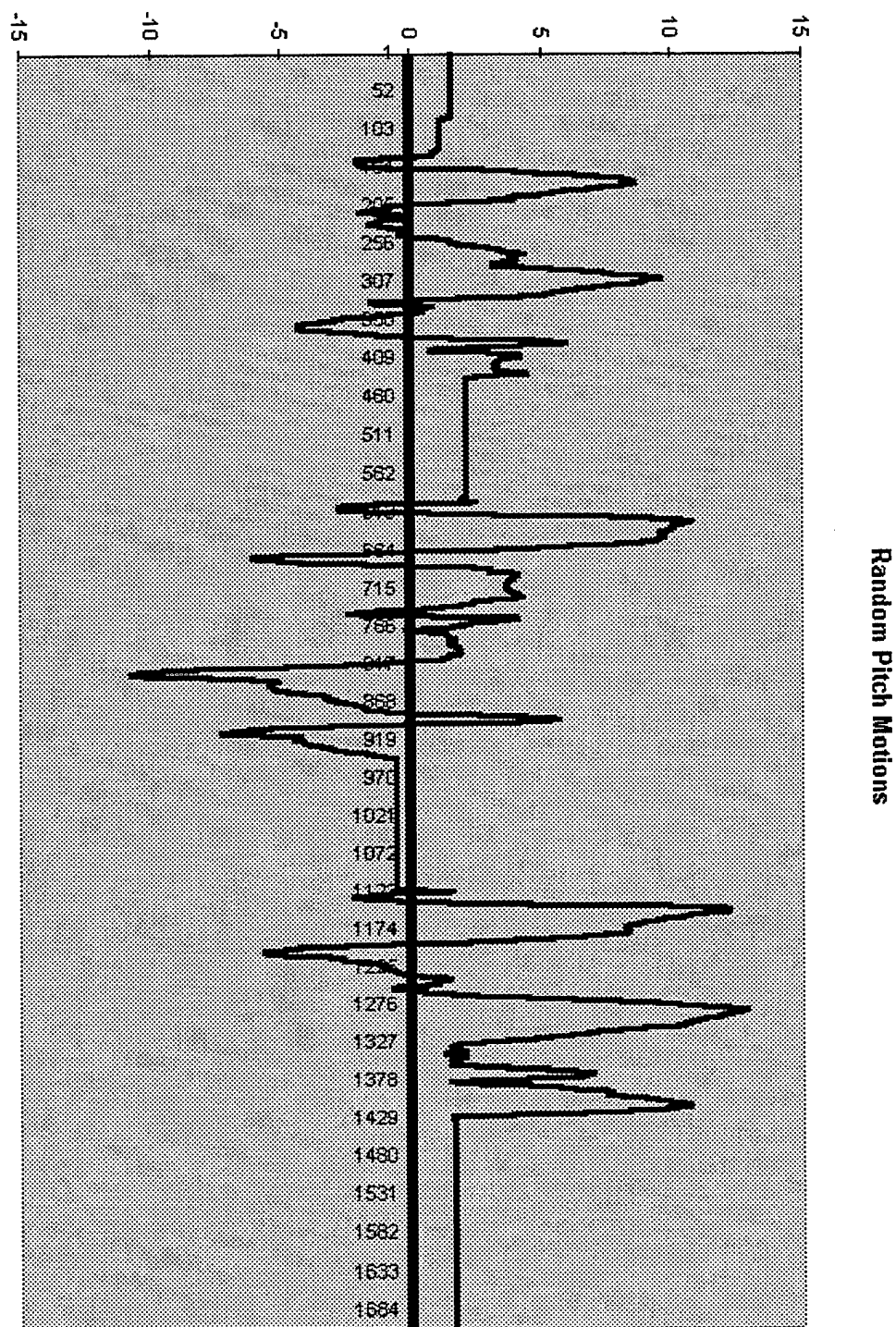


Figure 22: Pitch Angle for Random Motions.

sensor was rotated arbitrarily and placed flat on a table and then rotated again. This was repeated several times to simulate the random motions that may be encountered when tracking a human within a VE. The figures clearly show that when the sensor was placed in its reference position, flat on the table, it would report erroneous orientation data. This is shown by the flat line portions of the graphs which represent no rotations being applied to the sensor. Clearly, these lines are not at their reference position as they should be when the sensor is at rest. These results are concurrent with the performance observed when using the manufacturers graphical software. The error corrections being applied to the sensor output could not keep up with the rapid random rotations being applied. Since tracking human limb segments will doubtlessly encounter such random motions, for such applications the sensor needs to be re-configured to allow the application of an alternative filtering technique such as the quaternion filter presented in this thesis.

B. QUATERNION FILTER

The results of the tests performed on the implementation of the quaternion filter were better than expected. The filter performed extremely well under extreme error conditions. Deviations up to 90° were corrected by the filter. Applying the Gauss-Newton iteration method provided quadratic convergence to the correct orientation quaternion, in some cases in only one iteration. Overall, the thesis was a success, in that the theory for the quaternion filter was proven correct and an implementation of the filter now exists which can be applied to any VE simulation incorporating inertial tracking sensors technology.

The next, and final chapter, presents some final thoughts and recommendations for future work relating to inertial tracking of limb segment angles.

VI. SUMMARY AND CONCLUSIONS

A. SUMMARY

In this thesis, a quaternion filter is presented as an alternative to filters using Euler angles. This is because Euler angle implementations could encounter singularities that make them undesirable for human body tracking applications. Specifically, filters designed to use Euler angles experience divide-by-zero errors when calculating estimates of orientation at elevation angles of $\pm 90^\circ$ (as shown in Figure 7). This limitation is critical in simulations requiring the tracking of human limb segments which often rotate through vertical orientations. The work of this thesis shows that the quaternion filter proposed by [MCGH96A] provides a sound and viable solution to the Euler angle singularity problem, while at the same time simplifying the required filter computations.

Two different quaternion filter convergence methods were implemented and tested. The gradient descent method and the Gauss-Newton method [MCGH67] were tested under extreme error conditions of up to 90° . In both cases, the filter was able to quickly and accurately correct to the appropriate orientation quaternion. The gradient descent method provided gradual convergence, usually needing dozens of steps, while the Gauss-Newton method converged quadratically, often requiring just a single iteration for small initial errors. The results of the initial tests provide a decisive demonstration of the quaternion filter's applicability to VE simulations utilizing inertial sensors for human body tracking.

The Angularis inertial tracker [INTE97] was investigated as a possible alternative to traditional tracking methods by mechanical, electromagnetic, and acoustic tracking

technologies. The Angularis was built specifically as an inertial tracker for HMD applications. It was one of the goals of this thesis to investigate the viability of extending the use of the Angularis inertial sensor to human limb segment tracking. Tilt-table tests resulted in very accurate readings when this tracker was given steady, consistent rotations (shown in Figures 18, 19, and 20). However, when tested by hand, using rapid random rotations and orientations, the system had difficulty returning to its reference position (shown in Figures 21 and 22). These results suggest that the Angularis inertial system, in its current configuration, would not perform as well as would be required for human limb segment tracking applications. However, it is the author's understanding that this limitation can be removed by the manufacturer at the request of users.

B. CONCLUSIONS

It has been shown that filters based on quaternions have significant advantages for human body tracking in comparison to the more common Euler angle approach to attitude estimation. Two different convergence methods were investigated and tested in implementing the quaternion filter. Although both yielded acceptable results, the Gauss-Newton method was shown to be superior. Resulting in quadratic convergence, Gauss-Newton appears to be the preferred convergence method for further investigations concerning quaternion filters.

The integration of the quaternion filter with the Angularis inertial system proved to be an impossibility. The system, as it currently exists, does not allow for the application of independent filtering software. Better filtering techniques are required if the sensor is to be applied to human limb segment tracking. It has been demonstrated that a filter using

quaternions would be a viable solution to the filtering deficiencies experienced in the Angularis tests. A coupling of both entities could result in a very competitive system.

C. FUTURE WORK

The quaternion filter still requires extensive testing under realistic conditions using actual inertial sensors implemented in a VE simulation. This thesis presented a simulation incorporating "perfect sensors". No attempt was made to reproduce noise levels similar to those present in real sensor output. Research of the effects of noise on the filter's performance is still needed and required. Also, the use of dynamic objects must be investigated. The current evaluation was conducted using a simulated static sensor with a constant orientation. This type of simulation does not realistically demonstrate the environment in which the filter will be required to operate. In order to be useful, the filter must be capable of performing under the dynamic conditions of a typical VE simulation. The Angularis sensor could also be further improved by reducing its current size. The Angularis is certainly an improvement over traditional inertial systems, but is still a bit too large for reasonable incorporation into a body suit capable of tracking an entire human body. It is the author's opinion that, notwithstanding the current limitations of the Angularis sensor, the cutting edge tracking systems of the future will incorporate similar inertial sensors with filtering being done by a quaternion filter similar to, if not the same as, the one presented in this thesis. It is hoped that the work of this thesis makes a significant contribution toward the realization of such systems.

APPENDIX A. DERIVATION OF X MATRIX

The X matrix is defined by (eq. 3.29) as

$$X_{ij}^T = \left[\frac{\partial y_i}{\partial \hat{q}_j} \right]_{4 \times 6}$$

The elements of the 4x6 matrix come from the partial derivatives of the components of the computed measurement vector, $\vec{y}(\hat{q})$, given by (eq. 3.24)

$$\vec{y}(\hat{q}) = (\hat{q}^{-1} m \hat{q}, \hat{q}^{-1} n \hat{q}) \quad \text{(eq. 3.24)}$$

So, taking the partial derivative of (eq. 3.24) with respect to q_0 , the result is

$$\frac{\partial \vec{y}}{\partial \hat{q}_0} = \frac{\partial}{\partial \hat{q}_0} (\hat{q}^{-1} m \hat{q}, \hat{q}^{-1} n \hat{q}) \quad \text{(eq. A.1)}$$

Applying the product rule to (eq. A.1), it follows that

$$\frac{\partial \vec{y}}{\partial \hat{q}_0} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_0} m \hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_0}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_0} n \hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_0} \right) \quad \text{(eq. A.2)}$$

where

$$\frac{\partial q}{\partial q_0} = (1 \ 0 \ 0 \ 0) \quad \text{(eq. A.3)}$$

and

$$\frac{\partial q^{-1}}{\partial q_0} = (1 \ 0 \ 0 \ 0) \quad \text{(eq. A.4)}$$

Likewise, for q_1 , q_2 , and q_3 .

$$\frac{\partial \vec{y}}{\partial \hat{q}_1} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_1} m \hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_1}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_1} n \hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_1} \right) \quad (\text{eq. A.5})$$

$$\frac{\partial \vec{y}}{\partial \hat{q}_2} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_2} m \hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_2}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_2} n \hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_2} \right) \quad (\text{eq. A.6})$$

$$\frac{\partial \vec{y}}{\partial \hat{q}_3} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_3} m \hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_3}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_3} n \hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_3} \right) \quad (\text{eq. A.7})$$

where the corresponding quaternion partial derivatives are

$$\frac{\partial q}{\partial q_1} = (0 \ 1 \ 0 \ 0) \quad (\text{eq. A.8})$$

$$\frac{\partial q}{\partial q_2} = (0 \ 0 \ 1 \ 0) \quad (\text{eq. A.9})$$

$$\frac{\partial q}{\partial q_3} = (0 \ 0 \ 0 \ 1) \quad (\text{eq. A.10})$$

and, the partial derivatives of the inverse are given by

$$\frac{\partial q^{-1}}{\partial q_1} = (0 \ -1 \ 0 \ 0) \quad (\text{eq. A.11})$$

$$\frac{\partial q^{-1}}{\partial q_2} = (0 \ 0 \ -1 \ 0) \quad (\text{eq. A.12})$$

$$\frac{\partial q^{-1}}{\partial q_3} = (0 \ 0 \ 0 \ -1) \quad (\text{eq. A.13})$$

The partial derivatives as defined in (eq. A.2), (eq. A.5), (eq. A.6), and (eq. A.7) result in the partial derivatives of the m and n vectors with respect to q_0, q_1, q_2 , and q_3 , respectively.

Taking the results computed above, the $X_{6 \times 4}$ matrix can be constructed as follows

$$X = \begin{bmatrix} \frac{\partial \vec{y}}{\partial \hat{q}_0} & \frac{\partial \vec{y}}{\partial \hat{q}_1} & \frac{\partial \vec{y}}{\partial \hat{q}_2} & \frac{\partial \vec{y}}{\partial \hat{q}_3} \end{bmatrix}_{6 \times 4} \quad (\text{eq. A.14})$$

Note that the partial derivatives are column vectors in the X matrix, and that the transpose of X is required when applied in the filter. Thus, (eq. 3.29) becomes

$$X^T = \begin{bmatrix} \frac{\partial \vec{y}}{\partial \hat{q}_0} \\ \frac{\partial \vec{y}}{\partial \hat{q}_1} \\ \frac{\partial \vec{y}}{\partial \hat{q}_2} \\ \frac{\partial \vec{y}}{\partial \hat{q}_3} \end{bmatrix}_{4 \times 6}$$

APPENDIX B. DERIVATION OF GRADIENT

In the filter the gradient of the error criterion function $\phi(\hat{q})$, is given by (eq. 3.32)

$$\nabla\phi(\hat{q}) = \left(\frac{\partial\phi}{\partial\hat{q}_0}, \frac{\partial\phi}{\partial q_1}, \frac{\partial\phi}{\partial q_2}, \frac{\partial\phi}{\partial q_3} \right)^T = 2X_{4 \times 6}^T \varepsilon(\hat{q})_{6 \times 1}$$

where $\phi(\hat{q})$ is given by (eq. 3.26)

$$\phi(\hat{q})_{1 \times 1} = \varepsilon^T(\hat{q})_{1 \times 6} \varepsilon(\hat{q})_{6 \times 1}$$

In order to simplify the derivation of the gradient, consider the case of 1 dimensional

orientation vectors, where the measured vector is $y_0 = (h_1)$ and the calculated vector

is $y(\hat{q}) = (\hat{h}_1)$. The error then becomes

$$\varepsilon(\hat{q}) = \hat{h}_1 - h_1 \quad (\text{eq. B.1})$$

and

$$\phi(\hat{q}) = \varepsilon^2 = (\hat{h}_1 - h_1)^2 = \hat{h}_1^2 - 2h_1\hat{h}_1 + h_1^2 \quad (\text{eq. B.2})$$

Taking the partial derivative of (eq. B.2) with respect to q_0 yields

$$\frac{\partial\phi}{\partial\hat{q}_0} = 2\hat{h}_1 \frac{\partial\hat{h}_1}{\partial\hat{q}_0} - 2h_1 \frac{\partial\hat{h}_1}{\partial\hat{q}_0} = 2 \frac{\partial\hat{h}_1}{\partial\hat{q}_0} (\hat{h}_1 - h_1) = 2 \frac{\partial\hat{h}_1}{\partial\hat{q}_0} \varepsilon(\hat{q}) \quad (\text{eq. B.3})$$

Likewise, for the partial derivatives with respect to q_1 , q_2 , and q_3 . Thus, extending this result to the 6 dimensional case and arranging the partial derivatives in matrix form yields (eq. 3.32).

APPENDIX C. SIMULATION MODEL CODE

File: test-filter.lsp

```
; positive offsets from the positive x-axis
(setf 0-degrees '(0 1 0 0))
(setf 1-degree '(0 0.9998477 0.017452406 0))
(setf 2-degrees '(0 0.99939083 0.034899497 0))
(setf 3-degrees '(0 0.99862953 0.052335956 0))
(setf 4-degrees '(0 0.99756405 0.069756474 0))
(setf 5-degrees '(0 .9961946981 .0871557427 0))
(setf 10-degrees '(0 .9848077530 .173648177 0))
(setf 20-degrees '(0 .939692621 .342020143 0))
(setf 30-degrees '(0 .8660254037 .5 0))
(setf 40-degrees '(0 .766044443119 .642787609687 0))
(setf 50-degrees '(0 .642787609687 .766044443119 0))
(setf 60-degrees '(0 .5 .8660254037 0))
(setf 70-degrees '(0 .342020143 .939692621 0))
(setf 80-degrees '(0 .173648177 .9848077530 0))
(setf 85-degrees '(0 .0871557427 .9961946981 0))
(setf 89-degrees '(0 .017452406 .999847695 0))
(setf 90-degrees '(0 0 1 0))

(defun error-difference (error)
  (apply #'+ (mapcar #'square error)))

(defun test-filter-gradient (k-value offset iterations error-threshold delta-t)
  (setf *k* k-value)
  (setf Sensor (make-instance 'inertial-sensor))
  (initialize Sensor)
  (calibrate-magnetometer Sensor -1 1 -1 1 -1 1)
  (let* ((accelerometer (take-accelerometer-reading Sensor))
        (magnetometer (take-magnetometer-reading Sensor))
        (angular-rate (take-angular-rate-sensor-reading Sensor))
        (q-hat offset)
        (delta-t delta-t)
        (error 1))
    (format t "~%initial q-hat ~A ~%" q-hat)
    (do ((x 1 (1+ x)))
        ((> x iterations))
      (format t "~%~%***** Iteration ~A *****~%" x)
      (if (>= error error-threshold)
          (setf output (quaternion-filter-gradient accelerometer
                                                    magnetometer
                                                    angular-rate
                                                    q-hat
                                                    delta-t)
                  q-hat (firstn 4 output)
                  error (error-difference (lastn 6 output))))))
```



```

    (setf x (1+ iterations))))))

(defun test-filter-gauss-newton (multiplier offset iterations error-threshold delta-t)
  (setf Sensor (make-instance 'inertial-sensor))
  (initialize Sensor)
  (calibrate-magnetometer Sensor -1 1 -1 1 -1 1)
  (let* ((accelerometer (take-accelerometer-reading Sensor))
        (magnetometer (take-magnetometer-reading Sensor))
        (angular-rate (take-angular-rate-sensor-reading Sensor))
        (q-hat offset)
        (delta-t delta-t)
        (error 1))
    (format t "~%initial q-hat ~A ~%" q-hat)
    (do ((x 1 (1+ x)))
        ((> x iterations))
      (format t "~%~%***** Iteration ~A *****~%" x)
      (if (>= error error-threshold)
          (setf output (quaternion-filter-gauss-newton multiplier
                                                         accelerometer
                                                         magnetometer
                                                         angular-rate
                                                         q-hat
                                                         delta-t)
                  q-hat (firstn 4 output)
                  error (error-difference (lastn 6 output)))
          (setf x (1+ iterations))))))

```

File: sensor.lsp

```
(defclass inertial-sensor (quaternion-rigid-body)
  ((accelerometer
    :initform '(0 0 0)
    :accessor accelerometer)
   (angular-rate-sensor
    :initform '(0 0 0)
    :accessor angular-rate-sensor)
   (magnetometer
    :initform (make-instance '3-axis-magnetometer)
    :accessor magnetometer)))

(defmethod take-accelerometer-reading ((inertial-sensor inertial-sensor))
  (let* ((u (first (velocity inertial-sensor)))
        (v (second (velocity inertial-sensor)))
        (w (third (velocity inertial-sensor)))
        (p (fourth (velocity inertial-sensor)))
        (q (fifth (velocity inertial-sensor)))
        (r (sixth (velocity inertial-sensor)))
        (u-dot (first (velocity-growth-rate inertial-sensor)))
        (v-dot (second (velocity-growth-rate inertial-sensor)))
        (w-dot (third (velocity-growth-rate inertial-sensor)))
        (q1 (fifth (posture inertial-sensor)))
        (q2 (sixth (posture inertial-sensor)))
        (q3 (seventh (posture inertial-sensor)))
        (body-gravity-vector (rotate-vector
                              (quaternion-inverse (orientation-quaternion
                                                    inertial-sensor)) *m*)))
    (ax (+ u-dot (* w q) (* -1 v r) (second body-gravity-vector)))
    (ay (+ v-dot (* u r) (* -1 w p) (third body-gravity-vector)))
    (az (+ w-dot (* v p) (* -1 u q) (fourth body-gravity-vector))))

  (setf (accelerometer inertial-sensor)
        (normalize-vector (list ax ay az)))
  (accelerometer inertial-sensor)))

(defmethod take-angular-rate-sensor-reading ((inertial-sensor inertial-sensor))
  (setf (angular-rate-sensor inertial-sensor)
        (cons (fourth (velocity inertial-sensor))
              (cons (fifth (velocity inertial-sensor))
                    (list (sixth (velocity inertial-sensor))))))
  (angular-rate-sensor inertial-sensor))

; Max and min raw output over all orientations.
(defmethod calibrate-magnetometer
  ((inertial-sensor inertial-sensor) xmin xmax ymin ymax zmin zmax)
  (setf (x-bias (magnetometer inertial-sensor)) (/ (+ xmax xmin) 2)
        (y-bias (magnetometer inertial-sensor)) (/ (+ ymax ymin) 2)
        (z-bias (magnetometer inertial-sensor)) (/ (+ zmax zmin) 2))
```

```

(x-scale-factor (magnetometer inertial-sensor)) (/ (- xmax xmin) 2)
(y-scale-factor (magnetometer inertial-sensor)) (/ (- ymax ymin) 2)
(z-scale-factor (magnetometer inertial-sensor)) (/ (- zmax zmin) 2)))

(defmethod normalize-magnetometer-measurement-vector
  ((inertial-sensor inertial-sensor))
  (let* ((vx (x-reading (magnetometer inertial-sensor)))
        (vy (y-reading (magnetometer inertial-sensor)))
        (vz (z-reading (magnetometer inertial-sensor)))
        (xbias (x-bias (magnetometer inertial-sensor)))
        (ybias (y-bias (magnetometer inertial-sensor)))
        (zbias (z-bias (magnetometer inertial-sensor)))
        (xscale (x-scale-factor (magnetometer inertial-sensor)))
        (yscale (y-scale-factor (magnetometer inertial-sensor)))
        (zscale (z-scale-factor (magnetometer inertial-sensor)))
        (wx (normalize-reading vx xbias xscale))
        (wy (normalize-reading vy ybias yscale))
        (wz (normalize-reading vz zbias zscale)))
    (normalize-vector (list wx wy wz))))

(defmethod take-magnetometer-reading ((inertial-sensor inertial-sensor))
  (let* ((q-inv (quaternion-inverse (orientation-quaternion inertial-sensor)))
        (reading (rotate-vector q-inv *n*)))
    (setf (x-reading (magnetometer inertial-sensor)) (second reading)
          (y-reading (magnetometer inertial-sensor)) (third reading)
          (z-reading (magnetometer inertial-sensor)) (fourth reading)
          (magnetic-normal-vector (magnetometer inertial-sensor))
            (normalize-magnetometer-measurement-vector inertial-sensor))
    (magnetic-normal-vector (magnetometer inertial-sensor))))

(defclass 3-axis-magnetometer (quaternion-rigid-body)
  ((x-reading :accessor x-reading)
   (y-reading :accessor y-reading)
   (z-reading :accessor z-reading)
   (x-bias :accessor x-bias)
   (y-bias :accessor y-bias)
   (z-bias :accessor z-bias)
   (x-scale-factor :accessor x-scale-factor)
   (y-scale-factor :accessor y-scale-factor)
   (z-scale-factor :accessor z-scale-factor)
   (magnetic-normal-vector :accessor magnetic-normal-vector)))

(defun normalize-reading (value bias scale-factor)
  (/ (- value bias) scale-factor))

```

File: quaternion-filter

```
(defun earth-magnetic-field-unit-vector (deviation dip-angle)
  (rest (rotate-vector (equivalent-quaternion deviation (- dip-angle) 0)
    '(0 1 0 0))))

;normalized earth magnetic field vector in earth coordinates for Monterey, CA.
(setf *n* (cons 0 (earth-magnetic-field-unit-vector (deg-to-rad 15) (deg-to-rad 60))))

;normalized gravity vector in earth coordinates.
(setf *m* '(0 0 0 1))

;estimated sensor bias in body coordinates.
(setf *angular-rate-sensor-bias* '(0 0 0))

(defun calculated-measurement-vector (quaternion)
  (append (rest (rotate-vector (quaternion-inverse quaternion) *m*))
    (rest (rotate-vector (quaternion-inverse quaternion) *n*)))))

(defun make-X-transpose-matrix (quaternion vector)
  (list (append (rest (partial-derivative-q0 quaternion (firstn 4 vector)))
    (rest (partial-derivative-q0 quaternion (cddddr vector))))
    (append (rest (partial-derivative-q1 quaternion (firstn 4 vector)))
    (rest (partial-derivative-q1 quaternion (cddddr vector))))
    (append (rest (partial-derivative-q2 quaternion (firstn 4 vector)))
    (rest (partial-derivative-q2 quaternion (cddddr vector))))
    (append (rest (partial-derivative-q3 quaternion (firstn 4 vector)))
    (rest (partial-derivative-q3 quaternion (cddddr vector))))))

; alternative way to compute partial derivatives
(defun make-X-transpose-matrix-2 (quaternion m n)
  (list (partial-derivative-q0-2 quaternion (append m n))
    (partial-derivative-q1-2 quaternion (append m n))
    (partial-derivative-q2-2 quaternion (append m n))
    (partial-derivative-q3-2 quaternion (append m n))))

(defun quaternion-filter-gradient (accelerometer
  magnetometer
  angular-rate
  q-hat delta-t)
  (let* ((measured-y (append accelerometer magnetometer))
    (calculated-y (calculated-measurement-vector q-hat))
    (error (vector-subtract calculated-y measured-y))
    (X-trans (make-X-transpose-matrix q-hat (append *m* *n*)))
    (gradient-phi (scalar-multiply-vector 2 (post-multiply X-trans error)))
    (K (scalar-multiply-matrix (* -1 *k*) (unit-matrix 4)))
    (omega (cons '0 (vector-subtract angular-rate *angular-rate-sensor-bias*)))
    (q-dot (scalar-multiply-vector 0.5 (quaternion-product q-hat omega)))
    (q-hat-dot (vector-add q-dot (post-multiply K gradient-phi)))
    (new-q-hat (normalize-vector (vector-add q-hat
      (scalar-multiply-vector delta-t q-hat-dot)))))
```

```

(format t "~%q-hat ~A ~%" q-hat)
(format t "~%measured-y ~A ~%" measured-y)
(format t "~%calculated-y ~A ~%" calculated-y)
(format t "~%error ~A ~%" error)
(format t "~%gradient-phi ~A ~%" gradient-phi)
(format t "~%new-q-hat ~A ~%" new-q-hat)
(append new-q-hat error)))

(defun quaternion-filter-gauss-newton (multiplier
                                       accelerometer
                                       magnetometer
                                       angular-rate
                                       q-hat delta-t)
  (let* ((measured-y (append accelerometer magnetometer))
        (calculated-y (calculated-measurement-vector q-hat))
        (error (vector-subtract calculated-y measured-y))
        (X-trans (make-X-transpose-matrix q-hat (append *m* *n*)))
        (X-squared-inv (matrix-inverse (matrix-multiply X-trans (transpose X-trans))))
        (delta-q (scalar-multiply-vector (* -1 multiplier)
                                           (post-multiply X-squared-inv (post-multiply X-trans error))))
        (omega (cons '0 (vector-subtract angular-rate *angular-rate-sensor-bias*)))
        (q-dot (scalar-multiply-vector 0.5 (quaternion-product q-hat omega)))
        (q-hat-dot (vector-add q-dot delta-q))
        (new-q-hat (normalize-vector (vector-add q-hat
                                                  (scalar-multiply-vector delta-t q-hat-dot)))))
    (format t "~%q-hat ~A ~%" q-hat)
    (format t "~%measured-y ~A ~%" measured-y)
    (format t "~%calculated-y ~A ~%" calculated-y)
    (format t "~%error ~A ~%" error)
    (format t "~%delta-q ~A ~%" delta-q)
    (format t "~%new-q-hat ~A ~%" new-q-hat)
    (append new-q-hat error)))

```

File: quat-rigid-body.lsp

```
(defconstant *gravity* 32.2185)
(defclass quaternion-rigid-body ()
  (
    ;the vector (xe ye ze q0 q1 q2 q3).
    (posture
     :initform '(0 0 0 0 1 0 0)
     :initarg :posture
     :accessor posture)

    ;the vector (xe-dot ye-dot ze-dot q0-dot q1-dot q2-dot q3-dot).
    (posture-rate
     :initarg :posture-rate
     :accessor posture-rate)

    ;the vector (u v w p q r) in body coordinates.
    (velocity
     :initform '(0 0 0 0 0 0)
     :initarg :velocity
     :accessor velocity)

    ;the vector (u-dot v-dot w-dot p-dot q-dot r-dot).
    (velocity-growth-rate
     :accessor velocity-growth-rate)

    ;the vector (Fx Fy Fz L M N) in body coordinates.
    (forces-and-torques
     :initform (list 0 0 (- *gravity*) 0 0 0)
     :accessor forces-and-torques)

    ;the vector (Ix Iy Iz) in principal axis coordinates.
    (moments-of-inertia
     :initform '(1 1 1)
     :initarg :moments-of-inertia
     :accessor moments-of-inertia)

    (mass
     :initform 1
     :initarg :mass
     :accessor mass)

    (orientation-quaternion
     :initform '(0 1 0 0)
     :accessor orientation-quaternion)

    (position-quaternion
     :initform '(0 0 0 0)
     :accessor position-quaternion)

    (time-stamp
```

```

:accessor time-stamp)

;(0 x y z) in body coordinates for each node.
(node-list
:iniform '((0 0 0 0) (0 5 5 0) (0 -5 5 0) (0 -5 -5 0) (0 5 -5 0))
:iniform :node-list
:accessor node-list)

(polygon-list
:iniform '((1 2 3 4))
:iniform :polygon-list
:accessor polygon-list)

;(x y z 1) in earth coordinates.
(transformed-node-list
:accessor transformed-node-list)))

(defmethod initialize ((body quaternion-rigid-body))
  (setf (transformed-node-list body)
    (mapcar #'(lambda (node-location) (append (rest node-location) '(1)))
      (node-list body)))
  (setf (velocity-growth-rate body) (update-velocity-growth-rate body))
  (setf (posture-rate body) (earth-velocity body))
  (setf (time-stamp body) (get-constant-delta-t body)))

(defmethod quaternion-move ((body quaternion-rigid-body) x y z q0 q1 q2 q3)
  (setf (posture body) (list x y z q0 q1 q2 q3))
  (setf (orientation-quaternion body) (list q0 q1 q2 q3))
  (setf (position-quaternion body) (append '(0) (list x y z)))
  (transform-node-list body))

(defmethod get-constant-delta-t ((body quaternion-rigid-body)) 0.1)

(defmethod get-delta-t ((body quaternion-rigid-body))
  (let* ((new-time (get-internal-real-time))
    (delta-t (/ (- new-time (time-stamp body)) 1000)))
    (setf (time-stamp body) new-time)
    delta-t))

(defmethod update-rigid-body ((body quaternion-rigid-body))
  (let* ((delta-t (get-constant-delta-t body))
    (update-posture body delta-t)
    (setf (orientation-quaternion body)
      (list (fourth (posture body)) (fifth (posture body))
        (sixth (posture body)) (seventh (posture body)))))
    (setf (position-quaternion body)
      (list 0 (first (posture body))
        (second (posture body)) (third (posture body)))))
    (transform-node-list body)
    (update-velocity body delta-t)
    (update-velocity-growth-rate body)))

```

```

(defmethod update-velocity-growth-rate ((body quaternion-rigid-body))
  (setf (velocity-growth-rate body)
    (multiple-value-bind
      (Fx Fy Fz L M N u v w p q r Ix Iy Iz)
      (values-list
        (append
          (forces-and-torques body)
          (velocity body)
          (moments-of-inertia body))))
    (list (+ (* v r) (* -1 w q) (/ Fx (mass body))
      (* -1 (second (rotate-vector
        (quaternion-inverse (orientation-quaternion body))
        (append '(0 0 0) (list *gravity*)))))
      (+ (* w p) (* -1 u r) (/ Fy (mass body))
        (third (rotate-vector
          (quaternion-inverse (orientation-quaternion body))
          (append '(0 0 0) (list *gravity*)))))
      (+ (* u q) (* -1 v p) (/ Fz (mass body))
        (fourth (rotate-vector
          (quaternion-inverse (orientation-quaternion body))
          (append '(0 0 0) (list *gravity*)))))
      (/ (+ (* (- Iy Iz) q r) L) Ix)
      (/ (+ (* (- Iz Ix) r p) M) Iy)
      (/ (+ (* (- Ix Iy) p q) N) Iz)))))

(defmethod update-velocity ((body quaternion-rigid-body) delta-t)
  (setf (velocity body)
    (vector-add (velocity body)
      (scalar-multiply-vector delta-t (velocity-growth-rate body)))))

(defmethod update-posture ((body quaternion-rigid-body) delta-t)
  (setf (posture-rate body) (earth-velocity body))
  (setf (posture body)
    (append (vector-add (firstn 3 (posture body))
      (scalar-multiply-vector delta-t (firstn 3 (posture-rate body))))
      (normalize-vector (vector-add (cdddr (posture body))
        (scalar-multiply-vector delta-t (cdddr (posture-rate body)))))))

(defmethod transform-node-list ((body quaternion-rigid-body))
  (setf (transformed-node-list body)
    (mapcar #'(lambda (node-location)
      (append (rest (vector-add
        (position-quaternion body)
        (rotate-vector (orientation-quaternion body)
          node-location)))
        '(1)))
      (node-list body))))

(defmethod earth-velocity ((body quaternion-rigid-body))
  (let* ((linear-velocity (append '(0) (firstn 3 (velocity body))))
    (rotational-velocity (append '(0) (cdddr (velocity body)))))

```



```

    (linear-earth-velocity
     (rotate-vector (orientation-quaternion body) linear-velocity))
    (rotational-earth-velocity (scalar-multiply-vector 0.5
      (quaternion-product (orientation-quaternion body) rotational-velocity))))
    (append linear-earth-velocity rotational-earth-velocity)))

(defmethod print-body-posture ((body quaternion-rigid-body))
  (format t "~,2,,,F ~,2,,,F ~,2,,,F ~,2,,,F ~,2,,,F ~,2,,,F ~%"
    (first (posture body)) (second (posture body)) (third (posture body))
    (fourth (posture body)) (fifth (posture body)) (sixth (posture body))
    (seventh (posture body))))

(defmethod print-body-orientation-quaternion ((body quaternion-rigid-body))
  (format t "~,2,,,F ~,2,,,F ~,2,,,F ~,2,,,F ~%"
    (first (orientation-quaternion body)) (second (orientation-quaternion body))
    (third (orientation-quaternion body)) (fourth (orientation-quaternion body))))

```

File: partial-derivative.lsp

```
(defun partial-derivative-q0 (quaternion vector)
  (let ((q quaternion)
        (q-inv (quaternion-inverse quaternion))
        (partial-q0 '(1 0 0 0))
        (partial-q0-inv '(1 0 0 0))
        (v vector))
    (vector-add (quaternion-product partial-q0-inv (quaternion-product v q))
                 (quaternion-product q-inv (quaternion-product v partial-q0)))))

; alternative method
(defun partial-derivative-q0-2 (quaternion vector)
  (let ((q0 (first quaternion))
        (q1 (second quaternion))
        (q2 (third quaternion))
        (q3 (fourth quaternion))
        (m1 (first vector))
        (m2 (second vector))
        (m3 (third vector))
        (n1 (fourth vector))
        (n2 (fifth vector))
        (n3 (sixth vector)))
    (list (+ (* 2 q0 m1) (* -2 q2 m3) (* 2 q3 m2))
          (+ (* 2 q0 m2) (* 2 q1 m3) (* -2 q3 m1))
          (+ (* 2 q0 m3) (* -2 q1 m2) (* 2 q2 m1))
          (+ (* 2 q0 n1) (* -2 q2 n3) (* 2 q3 n2))
          (+ (* 2 q0 n2) (* 2 q1 n3) (* -2 q3 n1))
          (+ (* 2 q0 n3) (* -2 q1 n2) (* 2 q2 n1)))))

(defun partial-derivative-q1 (quaternion vector)
  (let ((q quaternion)
        (q-inv (quaternion-inverse quaternion))
        (partial-q1 '(0 1 0 0))
        (partial-q1-inv '(0 -1 0 0))
        (v vector))
    (vector-add (quaternion-product partial-q1-inv (quaternion-product v q))
                 (quaternion-product q-inv (quaternion-product v partial-q1)))))

; alternative method
(defun partial-derivative-q1-2 (quaternion vector)
  (let ((q0 (first quaternion))
        (q1 (second quaternion))
        (q2 (third quaternion))
        (q3 (fourth quaternion))
        (m1 (first vector))
        (m2 (second vector))
        (m3 (third vector))
        (n1 (fourth vector))
        (n2 (fifth vector))
        (n3 (sixth vector)))
```

```

(list (+ (* 2 q1 m1) (* 2 q2 m2) (* 2 q3 m3))
      (+ (* 2 q0 m3) (* -2 q1 m2) (* 2 q2 m1))
      (+ (* -2 q0 m2) (* -2 q1 m3) (* 2 q3 m1))
      (+ (* 2 q1 n1) (* 2 q2 n2) (* 2 q3 n3))
      (+ (* 2 q0 n3) (* -2 q1 n2) (* 2 q2 n1))
      (+ (* -2 q0 n2) (* -2 q1 n3) (* 2 q3 n1))))))

(defun partial-derivative-q2 (quaternion vector)
  (let ((q quaternion)
        (q-inv (quaternion-inverse quaternion))
        (partial-q2 '(0 0 1 0))
        (partial-q2-inv '(0 0 -1 0))
        (v vector))
    (vector-add (quaternion-product partial-q2-inv (quaternion-product v q))
                (quaternion-product q-inv (quaternion-product v partial-q2)))))

; alternative method
(defun partial-derivative-q2-2 (quaternion vector)
  (let ((q0 (first quaternion))
        (q1 (second quaternion))
        (q2 (third quaternion))
        (q3 (fourth quaternion))
        (m1 (first vector))
        (m2 (second vector))
        (m3 (third vector))
        (n1 (fourth vector))
        (n2 (fifth vector))
        (n3 (sixth vector)))
    (list (+ (* -2 q0 m3) (* 2 q1 m2) (* -2 q2 m1))
          (+ (* 2 q1 m1) (* 2 q2 m2) (* 2 q3 m3))
          (+ (* 2 q0 m1) (* -2 q2 m3) (* 2 q3 m2))
          (+ (* -2 q0 n3) (* 2 q1 n2) (* -2 q2 n1))
          (+ (* 2 q1 n1) (* 2 q2 n2) (* 2 q3 n3))
          (+ (* 2 q0 n1) (* -2 q2 n3) (* 2 q3 n2)))))

(defun partial-derivative-q3 (quaternion vector)
  (let ((q quaternion)
        (q-inv (quaternion-inverse quaternion))
        (partial-q3 '(0 0 0 1))
        (partial-q3-inv '(0 0 0 -1))
        (v vector))
    (vector-add (quaternion-product partial-q3-inv (quaternion-product v q))
                (quaternion-product q-inv (quaternion-product v partial-q3)))))

; alternative method
(defun partial-derivative-q3-2 (quaternion vector)
  (let ((q0 (first quaternion))
        (q1 (second quaternion))
        (q2 (third quaternion))
        (q3 (fourth quaternion))
        (m1 (first vector))

```

```

(m2 (second vector))
(m3 (third vector))
(n1 (fourth vector))
(n2 (fifth vector))
(n3 (sixth vector)))
(list (+ (* 2 q0 m2) (* 2 q1 m3) (* -2 q3 m1))
      (+ (* -2 q0 m1) (* 2 q2 m3) (* -2 q3 m2))
      (+ (* 2 q1 m1) (* 2 q2 m2) (* 2 q3 m3))
      (+ (* 2 q0 n2) (* 2 q1 n3) (* -2 q3 n1))
      (+ (* -2 q0 n1) (* 2 q2 n3) (* -2 q3 n2))
      (+ (* 2 q1 n1) (* 2 q2 n2) (* 2 q3 n3))))

```

File: quaternion-algebra.lsp

```

(defun quaternion-product (Q Q1)
  (let ((w (first Q)) (x (second Q)) (y (third Q)) (z (fourth Q))
        (w1 (first Q1)) (x1 (second Q1)) (y1 (third Q1)) (z1 (fourth Q1)))
    (list (- (* w w1) (* x x1) (* y y1) (* z z1))
          (+ (* x w1) (* w x1) (- (* z y1) (* y z1))
            (+ (* y w1) (* z x1) (* w y1) (- (* x z1)))
            (+ (* z w1) (- (* y x1) (* x y1) (* w z1))))))

(defun quaternion-inverse (Q)
  (list (first Q) (- (second Q)) (- (third Q)) (- (fourth Q))))

(defun rotate-vector (unit-quaternion vector) ;Vector is quaternion with leading
  (let* ((q unit-quaternion) (v vector) ;element zero.
        (q-inv (quaternion-inverse q)))
    (quaternion-product q (quaternion-product v q-inv))))

```

File: euler-to-quat.lsp

```

(defun equivalent-quaternion (azimuth elevation roll)
  (quaternion-product (set-quaternion-azimuth azimuth)
    (quaternion-product (set-quaternion-elevation elevation)
      (set-quaternion-roll roll))))

(defun set-quaternion-azimuth (angle)
  (list (cos (/ angle 2)) 0 0 (sin (/ angle 2))))

(defun set-quaternion-elevation (angle)
  (list (cos (/ angle 2)) 0 (sin (/ angle 2)) 0))

(defun set-quaternion-roll (angle)
  (list (cos (/ angle 2)) (sin (/ angle 2)) 0 0))

```

File: support-functions.lsp

```

(defun deg-to-rad (angle) (* 0.017453292519943295 angle))

(defun firstn (n list)
  (cond ((zerop n) nil)
        (t (cons (first list) (firstn (1- n) (rest list))))))

(defun lastn (n list)
  (cond ((zerop n) nil)
        (t (append (lastn (1- n) (firstn (1- (length list)) list)) (last list)))))

```

```
; this function not required for Allegro CL for Windows 95
;(defun square (value)
;  (* value value))
```

File: vector-matrix-arithmetic.lsp

```
(defun augment (matrix)
  (concat-matrix matrix (unit-matrix (length matrix))))

(defun concat-matrix (A B) ;A and B are matrices with equal number of rows.
  (cond ((null A) B)
        (t (cons (append (car A) (car B)) (concat-matrix (cdr A) (cdr B))))))

(defun cycle-left (matrix) (mapcar 'row-cycle-left matrix))

(defun cycle-up (matrix) (append (cdr matrix) (list (car matrix))))

(defun dot-product (vector-1 vector-2)
  (apply '+ (mapcar '* vector-1 vector-2)))

(defun first-square (matrix) ;Returns leftmost square matrix from argument.
  (do ((size (length matrix))
      (remainder matrix (rest remainder))
      (answer nil (cons (firstn size (first remainder)) answer)))
      ((null remainder) (reverse answer))))

(defun matrix-inverse (M)
  (do ((M1 (max-car-first (augment M))
      (cond ((null M1) nil) ;Abort for singular matrix.
            (t (max-car-firstn n (cycle-left (cycle-up M1))))))
      (n (1- (length M)) (1- n)))
      ((or (minusp n) (null M1)) (cond ((null M1) nil) (t (first-square M1))))
      (setq M1 (cond ((zerop (caar M1)) nil) (t (solve-first-column M1))))))

(defun matrix-multiply (matrix1 matrix2)
  (cond ((null (rest matrix1)) (list (pre-multiply (first matrix1) matrix2)))
        (t (cons (pre-multiply (first matrix1) matrix2)
                    (matrix-multiply (rest matrix1) matrix2)))))

(defun max-car-first (L) ;L is a list of lists. This function finds list with
  (cond ((null (cdr L)) L) ;largest car and moves it to head of list of lists.
        (t (if (> (abs (caar L)) (abs (caar (max-car-first (cdr L))))) L
                (append (max-car-first (cdr L)) (list (car L))))))

(defun max-car-firstn (n list)
  (append (max-car-first (firstn n list)) (nthcdr n list)))

(defun normalize-row (row) (scalar-multiply-vector (/ 1.0 (car row)) row))
```

```

(defun normalize-vector (vector)
  (scalar-multiply-vector (/ 1 (norm vector)) vector))

(defun norm (vector)
  (sqrt (apply #'+ (mapcar 'square vector)))))

(defun post-multiply (matrix vector)
  (cond ((null (rest matrix)) (list (dot-product (first matrix) vector)))
        (t (cons (dot-product (first matrix) vector)
                   (post-multiply (rest matrix) vector)))))

(defun pre-multiply (vector matrix)
  (post-multiply (transpose matrix) vector))

(defun row-cycle-left (row) (append (cdr row) (list (car row))))

(defun scalar-multiply-vector (scalar vector)
  (cond ((null vector) nil)
        (t (cons (* scalar (first vector))
                   (scalar-multiply-vector scalar (rest vector))))))

(defun scalar-multiply-matrix (scalar matrix)
  (if (not (null matrix))
      (cons (scalar-multiply-vector scalar (first matrix))
            (scalar-multiply-matrix scalar (rest matrix)))))

(defun solve-first-column (matrix) ;Reduces first column to (1 0 ... 0).
  (do* ((remaining-row-list matrix (rest remaining-row-list))
        (first-row (normalize-row (first matrix)))
        (answer (list first-row)
                 (cons (vector-add (first remaining-row-list)
                                   (scalar-multiply-vector (- (caar remaining-row-list)
                                                                first-row))
                                                                first-row))
                       answer)))
    ((null (rest remaining-row-list)) (reverse answer))))

(defun transpose (matrix) ;A matrix is a list of row vectors.
  (cond ((null (cdr matrix)) (mapcar 'list (car matrix)))
        (t (mapcar 'cons (car matrix) (transpose (cdr matrix))))))

(defun unit-vector (one-column length) ;Column count starts at 1.
  (do ((n length (1- n))
        (vector nil (cons (cond ((= one-column n) 1) (t 0)) vector)))
    ((zerop n) vector)))

(defun unit-matrix (size)
  (do ((row-number size (1- row-number))
        (I nil (cons (unit-vector row-number size) I)))
    ((zerop row-number) I)))

```

```
(defun vector-add (vector-1 vector-2) (mapcar '+ vector-1 vector-2))
```

```
(defun vector-subtract (vector-1 vector-2) (mapcar '- vector-1 vector-2))
```


APPENDIX D. SIMULATION MODEL TEST RUNS USING GRADIENT DESCENT METHOD

; (test-filter k offset max-iterations error delta-t)
(test-filter-gradient .7 10-degrees 100 .001 .1)

(-0.0407968594779338 0.987814785044294 0.149793422179633 -0.0109314855508384)
(-0.0359642597292573 0.989548859847923 0.139308154984263 -0.00963659435083765)
(-0.0333988632303714 0.991038020142009 0.129027401800444 -0.00894919843069551)
(-0.0309292427815184 0.992317459265856 0.119502133414506 -0.00828746562581398)
(-0.028640593210504 0.993416060803329 0.110658722418457 -0.00767422382150284)
(-0.0265169460797078 0.994358901419598 0.10245361366174 -0.00710519428779732)
(-0.0245474425832684 0.995167708988955 0.0948440349768008 -0.00657746741643614)
(-0.0227215761042829 0.995861275313425 0.0877894286954169 -0.00608822796790491)
(-0.0210294198307998 0.996455827433058 0.0812514389056478 -0.00563481606095787)
(-0.0194616178249921 0.996965358929859 0.0751939171141162 -0.00521472477960978)
(-0.0180093776235814 0.997401923540244 0.0695828918480387 -0.00482559819042578)
(-0.0166644560347113 0.997775893916602 0.0643865138599456 -0.00446522753680484)
(-0.0154191404136762 0.998096188622346 0.0595749837789895 -0.00413154622182663)
(-0.014266226825664 0.998370470494064 0.0551204677384272 -0.0038226239569759)
(-0.0131989962170749 0.998605319423938 0.0509970053086573 -0.00353666037726664)
(-0.012211189464283 0.998806382453515 0.0471804130930396 -0.00327197835557808)
(-0.0112969819654345 0.9989785038606 0.0436481865581401 -0.00302701719454714)
(-0.010450958277847 0.999125837688725 0.0403794020401677 -0.00280032583068047)
(-0.00966808717378177 0.999251944930007 0.0373546203678796 -0.00259055615056851)

; (test-filter k offset max-iterations error delta-t)
(test-filter-gradient .8 10-degrees 100 .001 .1)

(-0.0466363024932462 0.988054619864193 0.146345377276698 -0.0124961595910389)
(-0.0336448060715241 0.990016369242001 0.13658090515966 -0.0090150986163666)
(-0.0325765093727223 0.991635692032407 0.124584238621996 -0.00872884937864585)
(-0.0294805106677434 0.992994225320679 0.114153271356649 -0.00789927902587894)
(-0.0270504960876294 0.994134027077225 0.104466602633617 -0.00724815858154155)
(-0.0247418690563448 0.995089762769567 0.0956046694999285 -0.00662956383288417)
(-0.0226411424543706 0.99589074670975 0.0874768239624286 -0.0060666758363666)
(-0.0207132620817334 0.996561746884026 0.080030255471763 -0.00555010184741466)
(-0.0189480025981257 0.997123653967104 0.0732093910107909 -0.00507710199435059)
(-0.0173314839560756 0.997594062141267 0.066963725303663 -0.0046439571296634)
(-0.0158516748037761 0.99798777087449 0.0612461657160251 -0.00424744346235262)
(-0.0144972763022684 0.998317215928941 0.0560131748731002 -0.00388453347764369)
(-0.01325788503816 0.998592837826888 0.0512245341425605 -0.00355243958931966)
(-0.0121239028963317 0.99882339562866 0.0468431637111396 -0.00324858999018542)
(-0.01108649410346 0.999016233289641 0.0428349239064057 -0.00297061714191452)
(-0.0101375328618482 0.999177505170899 0.0391684192251775 -0.00271634374357618)
(-0.00926955382915876 0.999312366554594 0.0358148057669039 -0.00248376946271993)

; (test-filter k offset max-iterations error delta-t)
(test-filter-gradient .9 10-degrees 100 .001 .1)

(-0.0524760351956661 0.988246613548474 0.142888723539894 -0.0140609112526659)
(-0.0294523105119354 0.99045708628517 0.134406258574449 -0.00789172281690377)
(-0.0328759984859507 0.992198765059205 0.11992989262179 -0.00880909724467732)
(-0.0276080858352448 0.993618707837281 0.109109729514156 -0.00739756430412296)
(-0.0257098919446979 0.994781415767648 0.0984960848930969 -0.00688894478407322)
(-0.022999109006359 0.995733538414442 0.0891503444354995 -0.00616259268488926)
(-0.0208838215796623 0.996512849414438 0.0805896639495956 -0.00559580312714617)
(-0.0188515043342893 0.997150421192263 0.0728707295195703 -0.00505124536248464)
(-0.0170522400191014 0.997671831074489 0.0658730719325037 -0.00456913394225991)
(-0.0154104027156919 0.998098107382105 0.0595452533353919 -0.00412920496270799)
(-0.0139299632575643 0.998446516862027 0.0538198603326016 -0.0037325224054596)
(-0.0125894270161801 0.998731222574884 0.0486422866214839 -0.00337332680215604)
(-0.011377821997732 0.998963831698321 0.0439603649476757 -0.00304867821591733)
(-0.0102822040612832 0.999153850020246 0.0397274419986532 -0.00275510827463285)
(-0.00929182516300677 0.999309057873637 0.0359008362893083 -0.00248973704863885)

; (test-filter k offset max-iterations error delta-t)
(test-filter-gradient 1.0 10-degrees 100 .001 .1)

(-0.0583152058497816 0.988390682282916 0.139423955287233 -0.0156255123139037)
(-0.0233841113470548 0.990849531450444 0.13278226421806 -0.00626575375116281)
(-0.0351423427807272 0.992718898589437 0.114828290272939 -0.00941636236823354)
(-0.0244014962468546 0.994191294500872 0.104620680327731 -0.00653836121345577)
(-0.0252759303508672 0.995363513062414 0.0925566594905315 -0.00677266512546018)
(-0.0208793028447936 0.996299393515018 0.0831881826398405 -0.00559459233578731)
(-0.0194982062305918 0.997046937933426 0.0740940470762447 -0.00522452861334252)
(-0.0170032438817051 0.997643944329682 0.0663075625701921 -0.00455600546681241)
(-0.0153929960071094 0.9981205821668 0.0591721834286807 -0.00412454084920038)
(-0.0136473996816812 0.9985010080441 0.0528782862760464 -0.00365680972349124)
(-0.0122381790122951 0.998804566699065 0.0472113471120067 -0.00327921018317201)
(-0.0109045963485031 0.999046739936555 0.0421682799130494 -0.00292187778536873)
(-0.00974982771925494 0.999239909185641 0.0376526202393971 -0.00261245846371697)
(-0.00870022237272814 0.999393970093261 0.0336238620118091 -0.00233121755874369)

; (test-filter k offset max-iterations error delta-t)
(test-filter-gradient 1.1 10-degrees 100 .001 .1)

(-0.064152962096264 0.988486763117717 0.135951570888611 -0.0171897343857584)
(-0.0154383501349454 0.991160049344576 0.131705359403706 -0.00413669345112755)
(-0.0402208972792998 0.993164779155011 0.109040605997087 -0.0107771569448435)
(-0.0184433682027599 0.994695655034773 0.10107410049485 -0.00494188561563934)
(-0.0272749736590781 0.99587624841727 0.0862157891375802 -0.00730830716553013)
(-0.0170203420463652 0.996791380025529 0.078079790666324 -0.00456058690622508)
(-0.0195615695694212 0.997502548657585 0.0676648876773959 -0.00524150676881165)
(-0.0143499320307392 0.998055873164189 0.0605290761776836 -0.00384505269907804)
(-0.0145360733162948 0.99848660371862 0.0528964487868246 -0.00389492910622076)
(-0.0116238695093196 0.998821956912581 0.0470093955888315 -0.00311460644794699)
(-0.011028608011913 0.99908305378144 0.0412636498263716 -0.00295510661043149)
(-0.00923422502702625 0.999286324444643 0.0365437914021331 -0.00247430313871896)

; (test-filter k offset max-iterations error delta-t)
(test-filter-gradient 1.2 10-degrees 100 .001 .1

(-0.0699884515727754 0.988534814043835 0.132472072408145 -0.0187533490784299)
(-0.00561498066974421 0.99134280914623 0.131170282979858 -0.00150452953597429)
(-0.0489546216571366 0.993459214850227 0.102323841510331 -0.0131173513388009)
(-0.00780925307707703 0.995055303108517 0.0989928322489017 -0.00209248305549306)
(-0.0344797715312819 0.996259806578048 0.0786904519560616 -0.00923882693701665)
(-0.00815760284656923 0.99716942915779 0.0747114800169442 -0.00218582309491202)
(-0.024468423836436 0.997856825885085 0.0603329617468896 -0.00655629440703547)
(-0.00759078677198126 0.998376615370229 0.0564125598165019 -0.0020339451854692)
(-0.0174963668517397 0.998769890812519 0.0461584631606423 -0.00468813736840233)
(-0.00663835602497839 0.999067597782413 0.0426228033683549 -0.00177874213596324)
(-0.0126030030050699 0.999293061991474 0.0352581434831337 -0.0033769644774155)
(-0.00558803870908517 0.999463881528935 0.032225477558732 -0.00149731045937329)

LIST OF REFERENCES

- [BACH96A] Bachmann, E., et. al., "Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)", *Symposium on Autonomous Underwater Vehicle Technology, IEEE AUV 96*, Monterey, California, June, 1996.
- [BACH96B] Bachmann, E., *Research Notes: Quaternion Attitude Filter*, Computer Science Department, Naval Postgraduate School, Monterey, California, 1996.
- [BADL93] Badler, N., et. al., *Simulating Humans: Computer Graphics Animation and Control*, Oxford University Press, New York, New York, 1993.
- [BROW92] Brown, R., Hwang, P., *Introduction to Random Signals and Applied Kalman Filtering, Second Edition*, John Wiley and Sons, Inc., New York, New York, 1992.
- [COOK92] Cooke, J., et. al., "NPSNET: Flight Simulation Dynamic Modeling Using Quaternions", *Presence: Teleoperators and Virtual Environments*, Fall, 1992, Volume 1, Number 4, pp. 404-420.
- [CRAI89] Craig, J., *Introduction to Robotics, Mechanics and Control, Second Edition*, Addison-Wesley Publishing Company, Menlo Park, California, 1989.
- [FENG96] Feng, J., "An RF/Inertial Head-Tracker", *Phase II Proposal, NAVAIR SBIR Topic No. N95-144*, November, 1996, p. 2.
- [FOX94] Foxlin, E., and Durlach, N., "An Inertial Head-Orientation Tracker with Automatic Drift Compensation for Use with HMD's", *VRST'94 - Virtual Reality Software and Technology*, Singapore, August, 1994.
- [FREY96A] Frey, W., et. al., "Off-the-Shelf, Real-Time, Human Body Motion Capture for Synthetic Environments", *Technical Report NPSCS-96-003*, Naval Postgraduate School, Monterey, California, June, 1996.
- [FREY96B] Frey, W., *Application of Inertial Sensors and Flux-Gate Magnetometer to Real-Time Human Body Motion Capture*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, California, September, 1996.
- [GMD97] The German National Research Center for Information Technology, *Virtual Working Environment for Scientists, Physicians, and Architects*, <http://viswiz.gmd.de/>, Internet, 1997.

- [HANC96] Hancock, D., *An Interactive Computer Graphics System with Force Feedback Display*, Final Report on Haptics Project, <http://www.best.com/~dennish/haptics/>, Hewlett Packard Labs, San Mateo, California, 1996.
- [INTE96] InterSense, *IS-300 Series User's Guide*, InterSense Incorporated, 1996.
- [INTE97] InterSense, Inc., <http://www.isense.com>, Internet, 1997.
- [LIPM90] Lipman Electronic Engineering Ltd., *V-Scope VS-100 Owner's Guide Rev. 1.3*, Cat. No. 050-32-002, 1990.
- [MCGH67] McGhee, R., "Some Parameter-Optimization Techniques", in *Digital Computer User's Handbook*, McGraw-Hill, 1967, pp.234-253.
- [MCGH93] McGhee, R., *CS-4314 Class Notes: Derivation of Body Angular Rates to Euler Angle Rates Relationship*, Computer Science Department, Naval Postgraduate School, Monterey, California, 1993.
- [MCGH95] McGhee, R., et. al., "An Experimental Study of An Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)", *Proceedings of the 9th International Symposium on Unmanned, Untethered Submersible Technology*, Durham, NH, September, 1995, pp. 1-15.
- [MCGH96A] McGhee, R., *Research Notes: A Quaternion Attitude Filter Using Angular Rate Sensors, Accelerometers, and a 3-Axis Magnetometer*, Computer Science Department, Naval Postgraduate School, Monterey, California, 1996.
- [MCGH96B] McGhee, R., *CS-4920 Class Notes: Derivation of SANS Filter Equations*, Computer Science Department, Naval Postgraduate School, Monterey, California, 1996.
- [MCGH96C] McGhee, R., *Research Notes: Estimation of Heading From a 3-Axis Magnetometer*, Computer Science Department, Naval Postgraduate School, Monterey, California, 1996.
- [MEYE92] Meyer, K., Applewhite, H. L., and Biocca, F. A., "A Survey of Position Trackers", *Presence: Teleoperators and Virtual Environments*, Spring, 1992, Volume 1, Number 2, pp. 173-200.
- [NRG97] NPS Research Group Homepage, <http://www-npsnet.cs.nps.navy.mil/npsnet/pics.html>, Internet, 1997.
- [PAUL90] Paul, R., Funda, J., Taylor, R., "On Homogeneous Transforms, Quaternions, and Computational Efficiency", *IEEE Transactions on Robotics and Automation*, June, 1990, Volume 6, Number 3, pp. 382-388.

- [POLH93] Polhemus, *3Space Fastrak User's Manual Revision F*, OPM3609-002C, November, 1993.
- [ROBE97] Roberts, R., *Implementation and Evaluation of an Integrated, Self-Contained GPS/INS Shallow-Water AUV Navigation System (SANS)*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, California, March, 1997.
- [SKOP96] Skopowski, P. F., *Immersive Articulation of the Human Upper Body in a Virtual Environment*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, California, December, 1996.

